EQUIVALENCE OF CAST REPRESENTATIONS IN GRADUAL TYPING

Lu, Kuang-Chen

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science in Computer Science

in the Department of Computer Science,

Indiana University

April 15

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Doctoral Committee

_____

Jeremy Siek, PhD

_____

Sam Tobin-Hochstadt, PhD

_____

_____

Acknowledgments

I wish to show my gratitude to my committee, Jeremy Siek and Sam Tobin-Hochstadt, for their continuous help since I started my journey in Gradual Typing. I especially thank Jeremy, who is also my advisor, for supporting me to explore in sidetracks. I also thank all other members of the gradual typing group. All of them have given critical comments that shape this work. Additional thanks go to Jeremy Siek and Andre Kuhlenschmidt for hours of stimulating discussions.

I would like to pay my special regards to Dan Friedman. His books stimulate my interest in the field of programming languages. Thanks to Dan and Weixi Ma for cares and training on teaching and writing.

Last but not least, I thank my partner Jingxin Lei and my parents for their love, support, and understanding.

Lu, Kuang-Chen

EQUIVALENCE OF CAST REPRESENTATIONS IN GRADUAL TYPING

Gradual typing aims at combining dynamic typing and static typing in one language to provide the advantages of both typing disciplines. Implementing gradual typing with reasonable overhead, however, is not trivial. Casts might accumulate in higher-order values and continuations, thus change the overall space complexity. Several space-efficient cast representations have been proposed, including coercions in normal forms, threesomes, and supercoercions. Drawing an interface between a cast representation and the remaining parts of language implementations would likely help to understand the relation between these cast representations and potentially new ones. I propose such an interface, Cast Abstract Data Type, in this thesis. The Cast ADT brings another benefit. Space-efficient cast representations shall be proved not changing the overall behavior of language implementations except saving space. Existing proofs typically go through a whole bisimulation proof between the standard and inefficient semantics and a space-efficient one. If a space-efficient semantics uses the Cast ADT, its behavior would depend solely on the properties of the interface functions. Thus a cast representation is correct if its interface functions work correctly. I present an abstract machine $\mathcal{S}(C)$ that uses Cast ADT to work with casts, a parameterized bisimulation proof, and two subsets of Cast ADT that specify the Lazy D and the Lazy UD blame strategies. I also introduce a new cast representation, hypercoercions, that is better than the known cast representations in some aspects.

Jeremy Siek, PhD

Sam Tobin-Hochstadt, PhD

Contents

Chapter 1

Introduction

Dynamic typing and static typing are both useful in software development. The former enables rapid prototyping, while the later provides machine-checked specification and hence increase the robustness of software.

Around 2006, several groups of researchers proposed ways to integrate dynamic typing and static typing, notably gradual typing [Siek and Taha, 2006], hybrid typing [Knowles and Flanagan, 2010], migratory typing [Tobin-Hochstadt and Felleisen, 2006], and multi-language interoperability [Gray et al., 2005, Matthews and Findler, 2007].

Gradual typing [Siek and Taha, 2006, Siek et al., 2015b] permits imprecise type annotations while enforces type soundness. Consequently, gradually typed languages embed both static code and fully dynamic code, and enable a smooth transition in a spectrum of type precision [Siek et al., 2015b]. We give a formal description of gradual typing in Chapter 2.

A key difference between a gradual type systems and an ordinary one is that the former check type *consistency* instead of type equality. Two types are consistent if they have no conflicting type information. When the type of an expression is consistent but not equal to the type of its context, there would be a runtime computation (i.e. cast) that store or check type information to enforce type soundness.

Unfortunately, straightforward implementations of casts impose significant runtime overheads that can change the asymptotic space complexity of a program [Herman et al., 2010]. For example, the following module define functions that check the parity of natural numbers. The terms are ordinary, while the type annotations are not – the output type of `even?` is imprecise.

```
(define (even? [n : Int]) : ⋆
  (if (= n 0) #t (odd? (- n 1))))
(define (odd? [n : Int]) : Bool
  (if (= n 0) #f (even? (- n 1))))
```

These functions look tail-recursive, so it is natural to expect that they consumes constant space. However, this is not the case in a straightforward implementation. Because of the imprecise type annotation, there is a cast after each of the "tail call", one from **Bool** to $\star$, the other goes the other way. A straightforward implementation would cast the output of the recursive call *before* returning it, which effectively breaks the tail recursion and thus changes the space complexity.

Continuation is not the only place where casts might accumulate. Consider a variant of the above program written in continuation-passing style. Here, both functions returns **Bool** so there is not pending cast in continuations. The continuation argument `k`, however, is wrapped in a cast in each recursive call because the type of `k` is less precise in `even?`.

```
(define (even? [n : Int] [k : (→ ⋆ Bool)]) : Bool
  (if (= n 0) (k #t) (odd? (- n 1) k)))
(define (odd? [n : Int] [k : (→ Bool Bool)]) : Bool
  (if (= n 0) (k #f) (even? (- n 1) k)))
```

A solution to this space-efficiency problem is representing cast information more concisely. There are several known space-efficient cast representations, with various strengths and weaknesses [Siek et al., 2015a, Siek and Wadler, 2010, Garcia, 2013, Kuhlenschmidt et al., 2018, Siek and Garcia, 2012, García-Pérez et al., 2014]. The current state of the art includes

- threesomes [Siek and Wadler, 2010, Garcia, 2013],
- supercoercions [Garcia, 2013], and
- coercions in normal form [Siek and Garcia, 2012, Siek et al., 2015a].

These systems compress casts using a compose operator. Threesomes and supercoercions are good for mechanized metatheory because their compose operators are structurally recursive, making

them easy to define in proof assistants such as Agda. In contrast, the coercions in normal form have compose operators that are not structurally recursive, which makes it more difficult to define in Agda, requiring what amounts to an explicit proof of termination. On the other hand, coercions in normal form are easier to understand than threesomes (with a strange labeled bottom type), and supercoercions (10 different kinds).

This paper presents a new cast representation (Chapter 7), named *hypercoercions*, that is good for both mechanized metatheory and good for implementation. The straightforward Agda translation of composition operator for hypercoercions is able to pass the termination check and hypercoercions are suggestive of a bit-level representation that minimizes the need for pointers and fits all first-order casts into 64 bits. We present two flavors of hypercoercions to support the two blame tracking strategies from the literature: Lazy D and Lazy UD [Siek et al., 2009]. With the Lazy D strategy, only **d**own casts (casts from $\star$ to some other types) are subject to blame. With the Lazy UD strategy, however, some **u**p casts are also blamable. We are interested in the D blame tracking strategy because it comes with a more straightforward notion of safe cast compared to Lazy UD [Siek et al., 2009], which is why Lazy D was chosen in the Grift compiler [Kuhlenschmidt et al., 2018]. We are also interested in Lazy UD because it plays a prominent role in the gradual typing literature [Wadler and Findler, 2009]. The Lazy UD hypercoercions were inspired by the supercoercions of [Garcia, 2013] (hence the name) and the Lazy D hypercoercions were inspired by the normal forms of [Siek and Garcia, 2012]. The semantics of casts can be lazy or eager [Siek et al., 2009]. In this paper, we focus on lazy cast strategies because we suspect that they are more efficient than eager strategies and because New et al. [2019] show that the eager strategies are incompatible with η-equivalence of functions.

Of course, an alternative cast representation must be proved correct. That is, a language implementation should give the same output no matter whether we are using a cast representation to optimize space consumption. This paper presents a general framework for proving correctness of

cast representations (Chapter 4) and, in particular, proves hypercoercions are correct (Chapter 7). We believe the framework is applicable to all known cast representations. For illustration, we apply the framework to Lazy UD coercions in normal form (Chapter 5).

To summarize, the primary contributions of this thesis are:

- a framework in Agda for proving the correctness of Lazy D or Lazy UD cast representations.
- a formal proof that Lazy UD coercions in normal form are correct
- hypercoercions, a new cast representation, which has a more compact memory representation in comparison to coercions in normal form.
- a formal proof that hypercoercions are correct

Chapter 2

Background

In this chapter I will first review gradual typing (Section 2.1). Then I will describe gradual typing in the setting of a programming language – Gradually Typed Lambda Calculus (GTLC) (Section 2.2), and the dynamic semantics of GTLC (Section 2.3). There is a space-efficiency problem in the standard dynamic semantics. We review the problem together with its solutions in (Section 2.4).

2.1 Gradual Typing

Gradual typing [Siek and Taha, 2006, Siek et al., 2015b] permits imprecise type annotations while enforcing type soundness. To allow imprecision in type annotations, a gradual version of this type system would add a dynamic type, typically written as $\star$ or `Dyn`, to the type syntax. Consider a conventional static type system and its gradual version.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \tau \to \tau \mid \tau \times \tau \\ \text{(Gradual) Types} & T & ::= & \star \mid \texttt{Int} \mid \texttt{Bool} \mid T \to T \mid T \times T \end{array}$$

Note: In theory, to define a gradual type system, researchers usually start with a static one, such as the Simply Typed Lambda Calculus, and gradualizing it [Cimini and Siek, 2016, Garcia et al., 2016]. In practice, however, gradual type systems are often designed by adding type annotations to script languages. For instance, TypeScript is based on JavaScript [Bierman et al., 2014] and Reticulated Python is based on Python [Vitousek et al., 2014].

Of course, typing judgments have to understand imprecise types. Conventional type checkers insist type equality or subtyping in places such as function applications and if-expressions. But a gradual type systems allow implicit conversion between *consistent* types. Intuitively, consistency means the types have no conflicting information. For example, $\texttt{Int} \to \star$ and $\star \to \texttt{Bool}$ are consistent, but $\texttt{Int} \to \star$ and $\texttt{Bool} \to \texttt{Bool}$ are not. To be more specific, the consistency relation $T \sim T$ for our

example type system is defined as follows:

$$\frac{}{\star \sim T} \qquad \frac{}{T \sim \star}$$

$$\frac{}{\texttt{Int} \sim \texttt{Int}} \qquad \frac{}{\texttt{Bool} \sim \texttt{Bool}} \qquad \frac{S_1 \sim S_2 \quad T_1 \sim T_2}{S_1 \to T_1 \sim S_2 \to T_2} \qquad \frac{S_1 \sim S_2 \quad T_1 \sim T_2}{S_1 \times T_1 \sim S_2 \times T_2}$$

A gradual type checker exploits type consistency to allow imprecise type annotations while ruling out obvious type errors. For example, in a function application, the function domain might be different to the type of the argument, as long as they are consistent:

$$\frac{\Gamma \vdash e_1 : S \to T \qquad \Gamma \vdash e_2 : S' \qquad S \sim S'}{\Gamma \vdash (e_1\ e_2) : T}$$

In Section 2.2, I will further describe how typing judgments allow imprecise type annotations in the setting of a concrete gradually typed programming language.

Gradual typing also promises type soundness. A value of type `Int` must be an integer and a value of type $\star \to$ `Bool` must be a (partial) function returning booleans. Enforcing type soundness at the cost of a reasonable space consumption, however, turns out to be critical [Takikawa et al., 2016] but challenging [Herman et al., 2010]. I will describe how the dynamic semantics enforce type soundness in Section 2.3, the space-efficiency problem and some solutions in Section 2.4.

2.2 Gradually Typed Lambda Calculus (GTLC)

In this section I review the Gradually Typed Lambda Calculus (GTLC) [Siek and Taha, 2006, Siek et al., 2015b]. I have only one base type, `Bool`, as in the remaining chapters, but adding other base types should be straightforward.

Fig. 2.1 defines the types and type consistency we will be using in the remaining chapters. Here I introduce a new nonterminal, pretypes, which stands for types that are not $\star$. This definition is isomorphic to the previous one but more convenient in formal proof, where case-spliting $\star$ and other types happens a lot. Ground types is a subset of pretypes whose component are all $\star$s. They play a prominent role in Lazy UD blame tracking. The definition of $T \sim T$ is also changed slightly. Here I

$$
\begin{array}{lrcl}
\text{Types} & T, S & ::= & \star \mid P \\
\text{Pre-types} & P, Q & ::= & \texttt{Bool} \mid T \to T \mid T \times T \\
\text{Ground types} & G, H & ::= & \texttt{Bool} \mid \star \to \star \mid \star \times \star
\end{array}
$$

$\boxed{T \sim T}$ Consistency

$$
\frac{}{\star \sim \star} \qquad \frac{}{\star \sim P} \qquad \frac{}{P \sim \star}
$$

$$
\frac{}{\texttt{Bool} \sim \texttt{Bool}} \qquad \frac{S_1 \sim S_2 \quad T_1 \sim T_2}{S_1 \to T_1 \sim S_2 \to T_2} \qquad \frac{S_1 \sim S_2 \quad T_1 \sim T_2}{S_1 \times T_1 \sim S_2 \times T_2}
$$

$\boxed{T \smile T}$ Shallow consistency

$$
\frac{}{\star \smile \star} \qquad \frac{}{\star \smile P} \qquad \frac{}{P \smile \star}
$$

$$
\frac{}{\texttt{Bool} \smile \texttt{Bool}} \qquad \frac{}{S_1 \to T_1 \smile S_2 \to T_2} \qquad \frac{}{S_1 \times T_1 \smile S_2 \times T_2}
$$

Figure 2.1: Gradual types and type consistency

am using three $\star$-related rules so that every consistency relation has a unique proof object, which is again for the convenience of formal proofs. Shallow consistency $T \smile T$ is essentially a non-recursive version of consistency. It plays a prominent role in dynamic semantics.

Fig. 2.2 defines the syntax and static semantics of the GTLC. Several kinds of expressions are annotated with blame labels, which typically contain debugging information (e.g. source code location) in practice. When the dynamic semantics discovers a type inconsistency at runtime, it reports a blame label. The expression typing relation $\Gamma \vdash e : T$ further explains why we need blame labels. In function applications, we want to make sure the type of the function can be viewed as a function ($T_1 \triangleright T_{11} \to T_{22}$), and that the argument is consistent with the function domain. These checking might fail at runtime, in which case it is important to provide debugging information. The type matching relation $T \triangleright P$ matches a type against a pretype. This relation enable programmers to use the dynamic type as $\star \to \star$ or other ground types. In if-expressions, we want to make sure

Syntax

$$
\begin{array}{lrcl}
\text{Blame Labels} & l & & \\
\text{Variables} & x, y, z & & \\
\text{Expressions} & e & ::= & x \mid \lambda x : S.\ e \mid (e\ e)^l \mid \texttt{cons}\ e\ e \mid (\texttt{car}\ e)^l \mid (\texttt{cdr}\ e)^l \\
 & & \mid & \texttt{true} \mid \texttt{false} \mid (\texttt{if}\ e\ e\ e)^l
\end{array}
$$

$\boxed{\Gamma \vdash_{GTLC} e : T}$ Expressions Typing

$$
\cdots \quad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1 \triangleright T_{11} \to T_{12} \quad T_2 \sim T_{11}}{\Gamma \vdash (e_1\ e_2)^l : T_{12}} \quad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad \Gamma \vdash e_3 : T_3 \quad T_1 \triangleright \texttt{Bool} \quad T_2 \sim T_3}{\Gamma \vdash (\texttt{if}\ e_1\ e_2\ e_3)^l : T_2 \sqcap T_3}
$$

$$
\frac{\Gamma \vdash e : T \quad T \triangleright T_1 \times T_2}{\Gamma \vdash (\texttt{car}\ e) : T_1} \quad \frac{\Gamma \vdash e : T \quad T \triangleright T_1 \times T_2}{\Gamma \vdash (\texttt{cdr}\ e) : T_2}
$$

$\boxed{T \triangleright P}$ Type Matching

$$
\frac{}{P \triangleright P} \quad \frac{}{\star \triangleright G}
$$

$\boxed{T \sqcap T{=}\mathrm{T}}$

$$
\frac{}{\star \sqcap \star = \star} \quad \frac{}{\star \sqcap P = P} \quad \frac{}{P \sqcap \star = P}
$$

$$
\frac{S_1 \sqcap S_2 = S \quad T_1 \sqcap T_2 = T}{(S_1 \to T_1) \sqcap (S_2 \to T_2) = S \to T} \quad \frac{S_1 \sqcap S_2 = S \quad T_1 \sqcap T_2 = T}{(S_1 \times T_1) \sqcap (S_2 \times T_2) = S \times T}
$$

Figure 2.2: Syntax and static semantics of the Gradually Typed Lambda Calculus (GTLC)

Syntax

$$
\begin{array}{lrcl}
\text{Blame Labels} & l & & \\
\text{Variables} & x, y, z & & \\
\text{Expressions} & e & ::= & x \mid \lambda x : S.\ e \mid e\ e \mid \texttt{cons}\ e\ e \mid \texttt{car}\ e \mid \texttt{cdr}\ e \\
& & \mid & e : S \stackrel{l}{\Longrightarrow} T \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ e\ e
\end{array}
$$

$\boxed{\Gamma \vdash_{CC} e : T}$ Expression Typing

$$
\ldots \quad \frac{\Gamma \vdash e : S \qquad S \sim T}{\Gamma \vdash (e : S \stackrel{l}{\Longrightarrow} T) : T}
$$

Figure 2.3: Syntax and static semantics of the Cast Calculus (CC).

the test expression can be viewed as a `Bool` and that the two branches are consistent. The type of the whole if-expression would be $T_2 \sqcap T_3$, read as the *meet* of T_2 and T_3. The meet partial function is well-defined when its inputs are consistent. It combines the two types to eliminate as much imprecision as possible. Other typing rules are straightforward.

2.3 Dynamic Semantics of GTLC and Cast Calculus

Researchers usually define the dynamic semantics of gradually typed languages by translation to an intermediate language with casts. Examples of these intermediate languages include the blame calculus [Wadler and Findler, 2009, Siek et al., 2015a] and other cast calculi [Siek et al., 2009]. In this section I first review a conventional cast calculus (CC), the compilation from GTLC to CC, and finally the dynamic semantics of CC.

Syntax and Static Semantics The syntax and static semantics are the same for the Lazy D and Lazy UD cast calculi. They are reviewed in Fig. 2.3. As usual, the important features are the cast expressions, $e : S \stackrel{l}{\Longrightarrow} T$, which are responsible for runtime type checking. Polarities of blame labels is not treated but are straightforward to incorporate. The syntax and static semantics is the same as that of [Siek et al., 2009] except for a few minor exceptions:

$\boxed{\Gamma \vdash e_{GTLC} \rightsquigarrow e_{CC} : T}$ Compilation to Cast Calculus

$$\cdots \quad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : T_2 \qquad T_1 \triangleright T_{11} \to T_{12}}{\Gamma \vdash (e_1\ e_2)^l \rightsquigarrow ((e_1' : T_1 \overset{l}{\Longrightarrow} T_{11} \to T_{12})\ (e_2' : T_2 \overset{l}{\Longrightarrow} T_{11})) : T_{12}}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : T_2 \qquad \Gamma \vdash e_3 \rightsquigarrow e_3' : T_3 \qquad T = T_2 \sqcap T_3}{\Gamma \vdash (\texttt{if}\ e_1\ e_2\ e_3)^l \rightsquigarrow (\texttt{if}\ (e_1' : T_1 \overset{l}{\Longrightarrow} \texttt{Bool})\ (e_2' : T_2 \overset{l}{\Longrightarrow} T)\ (e_3' : T_3 \overset{l}{\Longrightarrow} T)) : T}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : T \qquad T \triangleright T_1 \times T_2}{\Gamma \vdash (\texttt{car}\ e)^l \rightsquigarrow (\texttt{car}\ (e' : T \overset{l}{\Longrightarrow} T_1 \times T_2)) : T_1}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : T \qquad T \triangleright T_1 \times T_2}{\Gamma \vdash (\texttt{cdr}\ e)^l \rightsquigarrow (\texttt{cdr}\ (e' : T \overset{l}{\Longrightarrow} T_1 \times T_2)) : T_2}$$

Figure 2.4: Compile GTLC to CC

- We add product type.
- We consider only one base type, Boolean, for simplicity.
- We separate types into those with a type constructor at the top, the *pre-types*, versus the dynamic type $\star$ (a.k.a. `Dyn` or ?).

As usual, for every cast $e : S \overset{l}{\Longrightarrow} T$, the source type S and the target T must be consistent.

Cast Insertion The compilation from GTLC to CC is also known as *cast insertion*, which is reviewed in Fig. 2.4. I only show compilation rules that involve blame labels. In a nutshell, cast insertion wrap an expression with a cast if the expression is involved in a type matching or in a consistency check.

Dynamic Semantics The dynamic semantics of a cast calculus is typically defined with a reduction semantics. Here we use a CEK machine [Felleisen and Friedman, 1986] instead. One reason is that environments are easier to deal with than substitutions in formal proof. Also, a CEK machine is more convenient to use for the space-efficient semantics in Chapter 4 (recall that a major point of space-efficiency is about compressing continuations and that continuations are part of the CEK

machine states). So using a CEK machine for the cast calculi in this section makes it easier to prove the equivalent of these machines. Of course, one should prove that the abstract machine presented here is equivalent to the standard reduction semantics for cast calculi, but I have not yet done so.

Fig. 2.6 defines the transition relation of the CEK machine ($\mathcal{R}(B)$) and Fig. 2.5 gives a grammar for machine states s, including a definition of values v and value typing. This machine is parameterized over blame strategy (B), which ranges over Lazy D (Fig. 2.7) and Lazy UD (Fig. 2.7). The transitions involving casts are highlighted in red and described in more detail below. The other transitions are standard for a CEK machine for an extended simply typed lambda calculus. Recall that CEK machines involve two kinds of transitions, (1) those that dive further into an expression (looking for a redex) and push an entry onto the continuation, and (2) those that return a value to the current continuation and possibly perform a computation. Corresponding to (1) and (2), the machine state is either in an evaluating $\langle e, E, K\rangle$ or returning $\langle v, K\rangle$ configuration, respectively. Additionally, there are two halting configurations: $\Downarrow v$ halts with a value, while $\Uparrow l$ halts with a blame label. Observables include all value constructors and blame labels, as in [Siek and Garcia, 2012]. The function converting values to observables ($\mathit{observe}(v) = o$) is defined in the obvious way.

Let v range over values. The $\mathtt{Dyn}_I(v)$ stands for values casted to the dynamic type, as in [Wadler and Findler, 2009]. The inner value must be of an injectable type. The definition of injectable types depends on blame strategies: for the Lazy D strategy, every pre-type is injectable (Fig. 2.7); for the Lazy UD strategy, only ground types are injectable (Fig. 2.8). Other kinds of values are either Boolean constants, functions, casted functions, pairs, or casted pairs. In earlier version of this work, we were tempted to use a generic constructor for all casted value, and use value typing rules to restrict the cast such that its source and target are shallowly consistent pre-types. This approach unfortunately considers $\mathtt{true} : \mathtt{Bool} \stackrel{l}{\Longrightarrow} \mathtt{Bool}$ as a value.

The syntax $\Uparrow l$ occurs several times in the syntax definition, we shall view those nonterminals that have $\Uparrow l$ case as error monads. We use the symbol $\gg\!=$ as the monad bind operator, which is

$$
\begin{array}{llcl}
\text{Values} & v, u & ::= & \mathtt{Dyn}_I(v) \mid \mathtt{true} \mid \mathtt{false} \\
& & \mid & \langle \lambda x : S.\, e, E \rangle \mid v : S \to T \overset{l}{\Longrightarrow} S \to T \\
& & \mid & \mathtt{cons}\ v\ v \mid v : S \times T \overset{l}{\Longrightarrow} S \times T \\
\text{Injectable types} & I, J & ::= & \text{a subset of } P \text{ defined by the blame strategy } B \\
\text{Environments} & E & ::= & \text{a partial function } \{\langle x, v \rangle, \dots\} \\
\text{Frames} & F & ::= & \Box\ \langle e, E \rangle \mid v\ \Box \mid \Box : S \overset{l}{\Longrightarrow} T \mid \mathtt{car}\ \Box \\
& & \mid & \mathtt{cdr}\ \Box \mid \mathtt{cons}\ \Box\ \langle e, E \rangle \mid \mathtt{cons}\ v\ \Box \\
\text{Continuations} & K & ::= & \Box \mid [F]K \\
\text{States} & s & ::= & \langle e, E, K \rangle \mid \langle v, K \rangle \mid \Downarrow v \mid \Uparrow l \\
\text{Observables} & o & ::= & \mathtt{dynamic} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{function} \mid \mathtt{cons} \mid \Uparrow l \\
\text{Cast results} & r & ::= & v \mid \Uparrow l
\end{array}
$$

$\boxed{\vdash v : T}$ Value typing

$$
\cdots \qquad \frac{\vdash v : I}{\vdash \mathtt{Dyn}_I(v) : \star}
$$

$$
\frac{\vdash v : S_1 \to T_1}{\vdash (v : S_1 \to T_1 \overset{l}{\Longrightarrow} S_2 \to T_2) : S_2 \to T_2} \qquad \frac{\vdash v : S_1 \times T_1}{\vdash (v : S_1 \times T_1 \overset{l}{\Longrightarrow} S_2 \times T_2) : S_2 \times T_2}
$$

$\boxed{\mathit{proxy}(v, S \overset{l}{\Longrightarrow} T) = v}$

$$
\begin{array}{rcl}
\mathit{proxy}(v, \mathtt{Bool} \overset{l}{\Longrightarrow} \mathtt{Bool}) & = & v \\
\mathit{proxy}(v, S_1 \to T_1 \overset{l}{\Longrightarrow} S_2 \to T_2) & = & v : S_1 \to T_1 \overset{l}{\Longrightarrow} S_2 \to T_2 \\
\mathit{proxy}(v, S_1 \times T_1 \overset{l}{\Longrightarrow} S_2 \times T_2) & = & v : S_1 \times T_1 \overset{l}{\Longrightarrow} S_2 \times T_2
\end{array}
$$

Figure 2.5: Syntax of a CEK machine $\mathcal{R}(B)$ and auxiliary data structures.

defined as follows:

$$\begin{array}{rcl} (\Uparrow l) \gg\!= f & = & \Uparrow l \\ a \gg\!= f & = & f(a) \quad \text{otherwise} \end{array}$$

The transition relation $s \longmapsto s$ invokes the cast application function, noted as $\mathit{applyCast}(v, S \stackrel{l}{\Longrightarrow} T)$ to allow for the differences between blame strategies (Fig. 2.7 and Fig. 2.8). When evaluating a cast expression, the machine moves the cast to the continuation and evaluates the inner expression. To cast a value, the machine invokes the cast application function on the value. The cast result is either a value or an error ($\Uparrow l$). If the cast succeeds, the machine returns the value to the next continuation. If the cast fails, the machine halts with the blame label. To apply a casted function, the machine first casts the argument v, then applies the inner function u to the casted argument, and finally casts the return value. To take out the first part of a casted pair, the machine takes out the first part of the inner pair u and cast it. Taking out the second part is similar.

The reflexive transitive closure of reduction ($s \longmapsto^* s$) and evaluation ($\mathit{eval}(e) = o$) are standard [Felleisen and Flatt, 2007].

Definition 1 (Lazy D CEK Machine, $\mathcal{R}(\texttt{LD})$)**.** The Lazy D CEK machine, written $\mathcal{R}(\texttt{LD})$, is the CEK machine of Fig. 2.6 using the `LD` blame strategy defined in Fig. 2.7. We write the transition relations of this machine as $s \longmapsto_{\mathcal{R}(\texttt{LD})} s$ and $s \longmapsto^*_{\mathcal{R}(\texttt{LD})} s$ and the evaluator as $\mathit{eval}_{\mathcal{R}(\texttt{LD})}(e) = o$.

We conjecture that $\mathcal{R}(\texttt{LD})$ agrees with the Lazy D cast calculus of [Siek et al., 2009].

Proposition 1 ($\mathcal{R}(\texttt{LD})$ is deterministic)**.** *If* $s \longmapsto_{\mathcal{R}(\texttt{LD})} s_1$ *and* $s \longmapsto_{\mathcal{R}(\texttt{LD})} s_2$ *then* $s_1 = s_2$.

Next, we define the CEK Machine for Lazy UD. The only differences with respect to Lazy D are in the injectable types and in the definition of the $\mathit{applyCast}(v, S \stackrel{l}{\Longrightarrow} T)$ function, in which a cast whose source or target is the dynamic type $\star$ is always split into two casts that go through a ground type, that is, a type in which all sub-components are the dynamic type, such as $\star \to \star$.

Definition 2 (Lazy UD CEK Machine)**.** The Lazy UD CEK machine, written $\mathcal{R}(\texttt{LUD})$ is the CEK machine of Fig. 2.6 using the `LUD` blame strategy defined in Fig. 2.8. We write the transition

$\boxed{s \longmapsto_{\mathcal{R}(B)} s}$ Transition

$$
\begin{array}{rcl}
\langle x, E, K\rangle & \longmapsto & \langle E(x), K\rangle \\
\langle \lambda x : S.\ e, E, K\rangle & \longmapsto & \langle \langle \lambda x : S.\ e, E\rangle, K\rangle \\
\langle e_1\ e_2, E, K\rangle & \longmapsto & \langle e_1, E, [\Box\ \langle e_2, E\rangle]K\rangle \\
\langle \texttt{true}, E, K\rangle & \longmapsto & \langle \texttt{true}, K\rangle \\
\langle \texttt{false}, E, K\rangle & \longmapsto & \langle \texttt{false}, K\rangle \\
\langle \texttt{if}\ e_1\ e_2\ e_3, E, K\rangle & \longmapsto & \langle e_1, [\texttt{if}\ \Box\ \langle e_2, E\rangle\ \langle e_3, E\rangle]K\rangle \\
\langle \texttt{cons}\ e_1\ e_2, E, K\rangle & \longmapsto & \langle e_1, E, [\texttt{cons}\ \Box\ \langle e_2, E\rangle]K\rangle \\
\langle \texttt{car}\ e, E, K\rangle & \longmapsto & \langle e, E, [\texttt{car}\ \Box]K\rangle \\
\langle \texttt{cdr}\ e, E, K\rangle & \longmapsto & \langle e, E, [\texttt{cdr}\ \Box]K\rangle \\
\langle e : S \overset{l}{\Longrightarrow} T, E, K\rangle & \longmapsto & \langle e, E, [\Box : S \overset{l}{\Longrightarrow} T]K\rangle \\
\langle v, [\Box : S \overset{l}{\Longrightarrow} T]K\rangle & \longmapsto & \mathit{applyCast}(v, S \overset{l}{\Longrightarrow} T) \gg\!= \lambda u.\langle u, K\rangle \\
\langle v_1, [\Box\ \langle e_2, E\rangle]K\rangle & \longmapsto & \langle e_2, E, [v_1\ \Box]K\rangle \\
\langle v, [\langle \lambda x : S.\ e, E\rangle\ \Box]K\rangle & \longmapsto & \langle e, E[x := v], K\rangle \\
\langle v, [u : S_1 \to T_1 \overset{l}{\Longrightarrow} S_2 \to T_2\ \Box]K\rangle & \longmapsto & \langle v, [\Box : S_2 \overset{l}{\Longrightarrow} S_1][u\ \Box][\Box : T_1 \overset{l}{\Longrightarrow} T_2]K\rangle \\
\langle \texttt{true}, [\texttt{if}\ \Box\ \langle e_2, E\rangle\ \langle e_3, E\rangle]K\rangle & \longmapsto & \langle e_2, E, K\rangle \\
\langle \texttt{false}, [\texttt{if}\ \Box\ \langle e_2, E\rangle\ \langle e_3, E\rangle]K\rangle & \longmapsto & \langle e_3, E, K\rangle \\
\langle v_1, [\texttt{cons}\ \Box\ \langle e_2, E\rangle]K\rangle & \longmapsto & \langle e_2, E, [\texttt{cons}\ v_1\ \Box]K\rangle \\
\langle v_2, [\texttt{cons}\ v_1\ \Box]K\rangle & \longmapsto & \langle \texttt{cons}\ v_1\ v_2, K\rangle \\
\langle \texttt{cons}\ v_1\ v_2, [\texttt{car}\ \Box]K\rangle & \longmapsto & \langle v_1, K\rangle \\
\langle u : S_1 \times T_1 \overset{l}{\Longrightarrow} S_2 \times T_2, [\texttt{car}\ \Box]K\rangle & \longmapsto & \langle u, [\texttt{car}\ \Box][\Box : S_1 \overset{l}{\Longrightarrow} S_2]K\rangle \\
\langle \texttt{cons}\ v_1\ v_2, [\texttt{cdr}\ \Box]K\rangle & \longmapsto & \langle v_2, K\rangle \\
\langle u : S_1 \times T_1 \overset{l}{\Longrightarrow} S_2 \times T_2, [\texttt{cdr}\ \Box]K\rangle & \longmapsto & \langle u, [\texttt{cdr}\ \Box][\Box : T_1 \overset{l}{\Longrightarrow} T_2]K\rangle \\
\langle v, \Box\rangle & \longmapsto & \Downarrow v
\end{array}
$$

$\boxed{\mathit{eval}_{\mathcal{R}(B)}(e) = o}$ Evaluation

$$\frac{\langle e, \emptyset, \Box\rangle \longmapsto^* \Downarrow v}{\mathit{eval}(e) = \mathit{observe}(v)} \qquad \frac{\langle e, \emptyset, \Box\rangle \longmapsto^* \Uparrow l}{\mathit{eval}(e) = \Uparrow l}$$

Figure 2.6: Dynamic semantics of the cast calculi as a CEK machine $\mathcal{R}(B)$. The transitions that involve casts are highlighted in red.

Injectable types $I, J \quad ::= \quad P$

$\boxed{applyCast_{\texttt{LD}}(v, S \overset{l}{\Longrightarrow} T) = r}$

$$
\begin{array}{rcll}
applyCast(v, \star \overset{l}{\Longrightarrow} \star) & = & v & \\
applyCast(v, P \overset{l}{\Longrightarrow} \star) & = & \texttt{Dyn}_P(v) & \\
applyCast(\texttt{Dyn}_P(u), \star \overset{l}{\Longrightarrow} Q) & = & applyCast(u, P \overset{l}{\Longrightarrow} Q) & \\
applyCast(v, P \overset{l}{\Longrightarrow} Q) & = & proxy(v, P \overset{l}{\Longrightarrow} Q) & \text{if } P \frown Q \\
applyCast(v, P \overset{l}{\Longrightarrow} Q) & = & \Uparrow l & \text{if } P \not\frown Q
\end{array}
$$

Figure 2.7: The Lazy D blame strategy (LD)

relations of this machine as $s \longmapsto_{\mathcal{R}(\texttt{LUD})} s$ and $s \longmapsto^*_{\mathcal{R}(\texttt{LUD})} s$ and the evaluator as $eval_{\mathcal{R}(\texttt{LUD})}(e) = o$.

We conjecture that $\mathcal{R}(\texttt{LUD})$ agrees with the Lazy UD cast calculus of [Siek et al., 2009].

Proposition 2 ($\mathcal{R}(\texttt{LUD})$ is deterministic)**.** *If* $s \longmapsto_{\mathcal{R}(\texttt{LUD})} s_1$ *and* $s \longmapsto_{\mathcal{R}(\texttt{LUD})} s_2$ *then* $s_1 = s_2$.

2.4 The Space Efficiency Problem

In this section, I review the space-efficiency problem in gradual typing. This problem is first observed when Findler and Felleisen [2002] proposed using proxies to enforce the specification of higher-order functions. Herman et al. [2010] describe this problem in details in the context of gradual typing and propose the first solution to this problem.

Let's reconsider the even-odd program in the setting of $\mathcal{R}(B)$.

```
(define (even? [n : Int]) : ⋆
  (if (= n 0) #t (odd? (- n 1))))
(define (odd? [n : Int]) : Bool
  (if (= n 0) #f (even? (- n 1))))
(even? 6)
```

Injectable types $I, J \quad ::= \quad G$

$\boxed{applyCast_{\texttt{LUD}}(v, S \xRightarrow{l} T) = r}$

$$
\begin{array}{rcll}
applyCast(v, \star \xRightarrow{l} \star) & = & v & \\
applyCast(v, G \xRightarrow{l} \star) & = & \texttt{Dyn}_G(v) & \\
applyCast(v, P \xRightarrow{l} \star) & = & \texttt{Dyn}_G(proxy(v, P \xRightarrow{l} G)) & \text{if } G \sim P, G \neq P \\
applyCast(\texttt{Dyn}_G(u), \star \xRightarrow{l} H) & = & project(u, G, l, H) & \\
applyCast(\texttt{Dyn}_G(u), \star \xRightarrow{l} Q) & = & project(u, G, l, H) \gg\!= \lambda v.proxy(v, H \xRightarrow{l} Q) & \text{if } H \sim Q, H \neq Q \\
applyCast(v, P \xRightarrow{l} Q) & = & proxy(v, P \xRightarrow{l} Q) & \text{if } P \smile Q \\
applyCast(v, P \xRightarrow{l} Q) & = & \Uparrow l & \text{if } P \not\smile Q
\end{array}
$$

$\boxed{project(v, G, l, H) = r}$

$$
\begin{array}{rcll}
project(v, G, l, G) & = & v & \\
project(v, G, l, H) & = & \Uparrow l & \text{if } G \neq H
\end{array}
$$

Figure 2.8: The Lazy UD blame strategy (`LUD`)

To run this program with $\mathcal{R}(B)$, we shall assume that we have extra language constructs, such as global definition and arithmetic operators. But these assumptions won't hurt the validity of this example.

After a few transition steps to set up the global environment, we would reach a state where we are evaluating the expression `(even? 6)` in the empty continuation. Then we reach the first "tail call", where the continuation is extended with a cast $\texttt{Bool} \xRightarrow{l_1} \star$ for some label l_1 because `odd?` returns a Boolean while `even?` should return a $\star$. When we reach the next "tail call", the continuation has been extended with anther cast frame, which has a different label because it has a different source-code location. The continuation will be keep extending until n becomes 0.

$$
\begin{aligned}
& \langle \texttt{(even? 6)}, \{...\}, \square \rangle \\
\longmapsto^* \quad & \langle \texttt{(odd? (- n 1))}, \{n \mapsto 6, \ldots\}, [\square : \texttt{Bool} \xRightarrow{l_1} \star] \square \rangle \\
\longmapsto^* \quad & \langle \texttt{(even? (- n 1))}, \{n \mapsto 5, \ldots\}, [\square : \star \xRightarrow{l_2} \texttt{Bool}][\square : \texttt{Bool} \xRightarrow{l_1} \star] \square \rangle \\
\longmapsto^* \quad & \langle \texttt{(odd? (- n 1))}, \{n \mapsto 4, \ldots\}, [\square : \texttt{Bool} \xRightarrow{l_1} \star][\square : \star \xRightarrow{l_2} \texttt{Bool}][\square : \texttt{Bool} \xRightarrow{l_1} \star] \square \rangle \\
\longmapsto^* \quad & \ldots
\end{aligned}
$$

Although the functions look tail-recursive, running the program in $\mathcal{R}(B)$ consumes linear space!

Fortunately, there are solutions to this problem – we may represent cast information more concisely. In the last continuation, the first two frames ($\square : \texttt{Bool} \xRightarrow{l_1} \star$ and $\square : \star \xRightarrow{l_2} \texttt{Bool}$) are not useful because they cancel each other.

Three space-efficient cast representations have been developed, including coercions in normal form, threesomes, and supercoercions. For instance, if we use coercions in normal form, the reduction sequence would looks like

$$
\begin{aligned}
& \langle \texttt{(even? 6)}, \{...\}, \square \rangle \\
\longmapsto^* \quad & \langle \texttt{(odd? (- n 1))}, \{n \mapsto 6, \ldots\}, [\square : \texttt{Bool!}] \square \rangle \\
\longmapsto^* \quad & \langle \texttt{(even? (- n 1))}, \{n \mapsto 5, \ldots\}, [\square : \texttt{Bool?}^{l_2}; \texttt{Bool!}] \square \rangle \\
\longmapsto^* \quad & \langle \texttt{(odd? (- n 1))}, \{n \mapsto 4, \ldots\}, [\square : \texttt{Bool!}] \square \rangle \\
\longmapsto^* \quad & \ldots
\end{aligned}
$$

In this case the last continuation is the same as the first one. We even get rid of the label l_1 because the machine would not blame it in any circumstance.

Chapter 3

Related Works

Coercions in normal form Henglein [1994] introduces coercions to study compilation from dynamic languages to static ones. Herman et al. [2010] proposes using coercions in normal forms to represent casts in gradual typing and proves space efficiency. However, they didn't show the coercion-based semantics is equivalent to the standard (and type-based) one. Siek et al. [2015a] improves the Lazy UD flavor of coercions in normal forms and proves that Lazy UD coercions in normal form is correct, using two bisimulation between three reduction semantics. The first semantics is similar to $\mathcal{R}(\texttt{LUD})$, which bisimulates a reduction semantics using coercions (not necessarily in normal form). The last semantics uses coercions in normal form and bisimulates the coercions-based semantics.

Threesomes Siek and Wadler [2010] introduces Lazy UD threesomes and prove its correctness, using a bisimulation between two reduction semantics: one is similar to $\mathcal{R}(\texttt{LUD})$; the other uses Lazy UD threesomes and is similar to the machine in Chapter 4. Garcia [2013] introduces several other flavors of threesomes, including a Lazy D one. Their threesomes are derived from corresponding flavors of coercions in normal form, using correctness-preserving transformation. So the correctness of their threesomes relies on the corrections of the coercions-in-normal-form-based semantics defined in Herman et al. [2010], which have not been proved correct.

Parameterized cast calculi García-Pérez et al. [2014] presents a dynamic semantics for the Eager D blame strategy, using coercions. They claims that "the artefacts for coercions can be replaced by other artefacts implementing different dynamic semantics".

Chapter 4

A Framework for Proving Correctness of Cast Representations

All cast representation shall be proved correct, that is, besides reducing space consumption, using a cast representation should not affect the results of program execution. In this chapter I present a framework (written in Agda [Norell, 2008]) for proving the correctness of cast representations. The framework includes an abstract data type named Cast ADT, which specifies the operators of cast representations, a CEK machine $\mathcal{S}(C)$, which uses the cast operators implemented by C to work with casts, and theorems relating $\mathcal{S}(C)$ to $\mathcal{R}(B)$. Section 4.1 describes Cast ADT and the machine $\mathcal{S}(C)$. I conjecture that the machine $\mathcal{S}(C)$ is space-efficient provided that the cast representation C performs compression. in later chapters I will prove or conjecture that all cast representations defined in the literature are instances of this ADT. Section 4.2 defines two subset of Cast ADT, namely Lazy D Cast ADT and Lazy UD Cast ADT. As their names suggest, if C is a Lazy D Cast ADT, executing a program in $\mathcal{S}(C)$ would give the same result as in $\mathcal{R}(\texttt{LD})$. Similarly, if C is a Lazy UD Cast ADT, executing a program in $\mathcal{S}(C)$ would give the same result as in $\mathcal{R}(\texttt{LUD})$. The theorems relating $\mathcal{S}(C)$ and $\mathcal{R}(B)$ are proved in Section 4.3.

4.1 Cast Abstract Data Type and a Space-efficient CEK Machine

Fig. 4.1 defines the syntax of the space-efficient CEK machine ($\mathcal{S}(C)$). This machine is parameterized over Cast ADT C. The key differences between $\mathcal{S}(C)$ and the $\mathcal{R}(B)$ machine are the places where casts can accumulate (Section 2.4), that is, values and continuations.

Let u, v range over values. Recall that in $\mathcal{R}(B)$ we have two value constructors for functions and another two for pairs. That setup allows zero-or-more layers of casts at the top of higher-order values, which might lead to space inefficiency (Section 2.4). In $\mathcal{S}(C)$, we instead have exactly one layer of cast. Thus non-casted values in $\mathcal{R}(B)$ correspond to values in $\mathcal{S}(C)$ where casts are identities

Runtime casts	c, d	::=	a data type defined by the instance of Cast ADT
Values	v, u	::=	$\texttt{Dyn}_I(v) \mid \texttt{true} \mid \texttt{false}$
		$\mid$	$\langle \lambda x : S.\ e, E \rangle : c \to c \mid \texttt{cons}\ v\ v : c \times c$
Injectable types (Lazy D)	I, J	::=	P
Injectable types (Lazy UD)	I, J	::=	G
Environments	E	::=	a partial function $\{\langle x, v \rangle, \dots\}$
Frames	F	::=	$\Box\ \langle e, E \rangle \mid v\ \Box \mid \texttt{car}\ \Box \mid \texttt{cdr}\ \Box$
		$\mid$	$\texttt{cons}\ \Box\ \langle e, E \rangle \mid \texttt{cons}\ v\ \Box$
Continuations	K	::=	$[\Box : c]k$
Pre-continuations	k	::=	$\Box \mid [F]K$
States	s	::=	$\langle e, E, K \rangle \mid \langle v, K \rangle \mid \Downarrow v \mid \Uparrow l$
Cast results	r	::=	$v \mid \Uparrow l$

Figure 4.1: Syntax of the space-efficient machine $\mathcal{S}(C)$ and auxiliary data structures.

($id(T)$). The id functions is provided by the Cast ADT.

Let K ranges over continuations and k ranges over pre-continuations. Pre-continuations are like the continuations in $\mathcal{R}(B)$, but their top-most frame must not be a cast frames. A continuation is now a pre-continuation prefixed with a cast frame. Recall that continuations in $\mathcal{R}(B)$ have zero or more casts at the top. In $\mathcal{S}(C)$, however, every continuation has exactly one cast at the top. Continuations in $\mathcal{R}(B)$ that have no casts at the top correspond to continuations in $\mathcal{S}(C)$ whose casts are identities constructed by $id(T)$. Continuations in $\mathcal{R}(B)$ that have many casts at the top correspond to continuations in $\mathcal{S}(C)$ where those casts are composed by $c \fatsemi c$, another functions provided by the Cast ADT. Injectable types (I, J), environments ($\mathcal{E}$), states (s), and cast results (r) are the same as in $\mathcal{R}(B)$.

Value typing ($\vdash v : T$), continuation typing ($\vdash k : S \Longrightarrow T$ and $\vdash K : S \Longrightarrow T$), and environment typing ($\Gamma \vdash E$) are defined in the obvious way. They rely on cast typing $\vdash c : S \Longrightarrow T$, which shall be provided by the Cast ADT.

Definition 3 (Cast Abstract Data Type (Cast ADT))**.** A cast abstract data type is a set $Cast$, which is indexed by two types, with four operators and two properties of operators:

- $id(T) = c$: create an identity cast
- $c \fatsemi c = c$: compose casts
- $\lceil S \xRightarrow{l} T \rceil = c$: translate cast in source program to runtime casts
- $[\![c]\!](v) = r$: apply a cast to a value
- If $\vdash v : T$ then $[\![id(T)]\!](v) = v$
- If $\vdash v : T_1$ and $\vdash c : T_1 \Longrightarrow T_2$ and $\vdash d : T_2 \Longrightarrow T_3$ then $[\![c \fatsemi d]\!](v) = ([\![c]\!](v) \gg= [\![d]\!])$

We use the syntax $\vdash c : S \Longrightarrow T$ to mean c is in the set $Cast_{S,T}$.

Before moving to the transition relation of $\mathcal{S}(C)$, let's define two helper functions that manipulate continuations. The function *mkCont* construct a continuation from a pre-continuation by topping the pre-continuation with an identity cast. The function *extCont* composes c with the cast at the top of K by using the composition operator $\fatsemi$.

$$\begin{aligned} mkCont(k) &= [\Box : id(S)]k \quad \text{if } \vdash k : S \Longrightarrow T \\ extCont(c, [\Box : d]k) &= [\Box : (c \fatsemi d)]k \end{aligned}$$

Fig. 4.2 defines the transition relation $s \longmapsto_{\mathcal{S}(C)} s$. They rely on functions provided by C to work with casts. To construct a function, the machine fills the casts with identities. When evaluating a cast expression, the machine extends the continuation with cast $\lceil S \xRightarrow{l} T \rceil$. To return a value to a continuation, the machine first applies the top cast to the value. If the cast application fails, the machine halts with the blame label from the failure. Otherwise, the machine handles the pre-continuation with the function $\langle\langle v, k \rangle\rangle$. To apply a function, the machine applies the domain cast c to the argument v and evaluates the function body. The codomain cast d is merged into the continuation with *extCont*. When the machine constructs a pair, it fills the casts with *id*s, just like how it did for functions. To take out the first part of a pair, the machine applies the first pending cast on the first value and returns the result to the continuation. Taking out the second part of a pair is similar.

$\boxed{s \longmapsto_{\mathcal{S}(C)} s}$ Transition

$$
\begin{array}{rcl}
\langle x, E, K\rangle & \longmapsto & \langle E(x), K\rangle \\
\langle \lambda x : S.\ e, E, K\rangle & \longmapsto & \langle\langle \lambda x : S.\ e, E\rangle : id(S) \to id(T), K\rangle \\
 & & \text{if } \Gamma, S \vdash e : T \text{ and } \Gamma \vdash E \\
\langle e_1\ e_2, E, K\rangle & \longmapsto & \langle e_1, E, mkCont([\Box\ \langle e_2, E\rangle]K)\rangle \\
\langle \texttt{true}, E, K\rangle & \longmapsto & \langle \texttt{true}, K\rangle \\
\langle \texttt{false}, E, K\rangle & \longmapsto & \langle \texttt{false}, K\rangle \\
\langle \texttt{if}\ e_1\ e_2\ e_3, E, K\rangle & \longmapsto & \langle e_1, E, mkCont([\texttt{if}\ \Box\ \langle e_2, E\rangle\ \langle e_3, E\rangle]K)\rangle \\
\langle \texttt{cons}\ e_1\ e_2, E, K\rangle & \longmapsto & \langle e_1, E, mkCont([\texttt{cons}\ \Box\ \langle e_2, E\rangle]K)\rangle \\
\langle \texttt{car}\ e, E, K\rangle & \longmapsto & \langle e, E, mkCont([\texttt{car}\ \Box]K)\rangle \\
\langle \texttt{cdr}\ e, E, K\rangle & \longmapsto & \langle e, E, mkCont([\texttt{cdr}\ \Box]K)\rangle \\
\langle e : S \overset{l}{\Longrightarrow} T, E, K\rangle & \longmapsto & \langle e, E, extCont(\lceil S \overset{l}{\Longrightarrow} T\rceil, K)\rangle \\
\langle v, [\Box : c]k\rangle & \longmapsto & [\![c]\!](v) \ggg\!= \lambda u.\langle\langle u, K\rangle\rangle
\end{array}
$$

$\boxed{\langle\langle v, k\rangle\rangle = s}$

$$
\begin{array}{rcl}
\langle\langle v_1, [\Box\ \langle e_2, E\rangle]K\rangle\rangle & = & \langle e_2, E, mkCont([v_1\ \Box]K)\rangle \\
\langle\langle v, [(\langle\langle \lambda x : S.\ e, E\rangle\rangle : c \to d)\ \Box]K\rangle\rangle & = & [\![c]\!](v) \ggg\!= \lambda u.\ \langle e, E[x := u], extCont(d, K)\rangle \\
\langle\langle \texttt{true}, [\texttt{if}\ \Box\ \langle e_2, E\rangle\ \langle e_3, E\rangle]K\rangle\rangle & = & \langle e_2, E, K\rangle \\
\langle\langle \texttt{false}, [\texttt{if}\ \Box\ \langle e_2, E\rangle\ \langle e_3, E\rangle]K\rangle\rangle & = & \langle e_3, E, K\rangle \\
\langle\langle v_1, [\texttt{cons}\ \Box\ \langle e_2, E\rangle]K\rangle\rangle & = & \langle e_2, E, mkCont([\texttt{cons}\ v_1\ \Box]K)\rangle \\
\langle\langle v_2, [\texttt{cons}\ v_1\ \Box]K\rangle\rangle & = & \langle \texttt{cons}\ v_1\ v_2 : id(T_1) \times id(T_2), K\rangle \\
 & & \text{if } \vdash v_1 : T_1 \text{ and } \vdash v_2 : T_2 \\
\langle\langle \texttt{cons}\ v_1\ v_2 : c_1 \times c_2, [\texttt{car}\ \Box]K\rangle\rangle & = & [\![c_1]\!](v_1) \ggg\!= \lambda u.\langle u, K\rangle \\
\langle\langle \texttt{cons}\ v_1\ v_2 : c_1 \times c_2, [\texttt{cdr}\ \Box]K\rangle\rangle & = & [\![c_2]\!](v_2) \ggg\!= \lambda u.\langle u, K\rangle \\
\langle\langle v, \Box\rangle\rangle & = & \Downarrow v
\end{array}
$$

$\boxed{eval(e) = o}$ Evaluation

$$
\frac{\langle e, \emptyset, mkCont(\Box)\rangle \longmapsto^* \Downarrow v}{eval(e) = observe(v)} \qquad \frac{\langle e, \emptyset, \Box\rangle \longmapsto^* \Uparrow l}{eval(e) = \Uparrow l}
$$

Figure 4.2: Dynamic semantics of the cast calculi as a space-efficient CEK machine, $\mathcal{S}(C)$. The transitions that involve casts are highlighted in red.

Reflexive transitive closure of reduction ($s \longmapsto_{\mathcal{S}(C)} s$) and evaluation ($\mathit{eval}_{\mathcal{S}(C)}(e) = o$) are the same as in $\mathcal{R}(B)$.

Proposition 3 ($\mathcal{S}(C)$ is deterministic)**.** *If* $s \longmapsto_{\mathcal{S}(C)} s_1$ *and* $s \longmapsto_{\mathcal{S}(C)} s_2$ *then* $s_1 = s_2$.

4.2 Specifying good instances of Cast ADT

Ultimately, we want to claim that if a C satisfies a collection of properties, $\mathcal{S}(C)$ would be equivalent to $\mathcal{R}(\texttt{LD})$ or $\mathcal{R}(\texttt{LUD})$. Cast ADT has specified the behavior of identities and composition, so the only remaining operator is translation. Lazy D Cast ADT (Definition 4) and Lazy UD Cast ADT (Definition 5) specify the behavior of $\lceil S \overset{l}{\Longrightarrow} T \rceil$. Readers are encouraged to compare them with the corresponding *applyCast* functions (Fig. 2.7 and Fig. 2.8).

Definition 4 (Lazy D Cast ADT)**.** A Cast is a Lazy D Cast if:

1. $[\![\lceil \star \overset{l}{\Longrightarrow} \star \rceil]\!](v) = v$
2. $[\![\lceil P \overset{l}{\Longrightarrow} \star \rceil]\!](v) = \texttt{Dyn}_P(v)$
3. $[\![\lceil \star \overset{l}{\Longrightarrow} Q \rceil]\!](\texttt{Dyn}_P(u)) = [\![\lceil P \overset{l}{\Longrightarrow} Q \rceil]\!](u)$
4. $[\![\lceil \texttt{Bool} \overset{l}{\Longrightarrow} \texttt{Bool} \rceil]\!](v) = v$
5. $[\![\lceil S_1 \to T_1 \overset{l}{\Longrightarrow} S_2 \to T_2 \rceil]\!](\langle \lambda x : S.\ e, E \rangle : c \to d)$
   $= \langle \lambda x : S.\ e, E \rangle : (\lceil S_2 \overset{l}{\Longrightarrow} S_1 \rceil \fatsemi c) \to (d \fatsemi \lceil T_1 \overset{l}{\Longrightarrow} T_2 \rceil)$
6. $[\![\lceil S_1 \times T_1 \overset{l}{\Longrightarrow} S_2 \times T_2 \rceil]\!](\texttt{cons}\ v_1\ v_2 : c \times d)$
   $= \texttt{cons}\ v_1\ v_2 : (c \fatsemi \lceil S_1 \overset{l}{\Longrightarrow} S_2 \rceil) \times (d \fatsemi \lceil T_1 \overset{l}{\Longrightarrow} T_2 \rceil)$
7. $[\![\lceil P \overset{l}{\Longrightarrow} Q \rceil]\!](v) = \Uparrow l$
   if $P \not\sim Q$

Definition 5 (Lazy UD Cast ADT)**.** A Cast is a Lazy UD Cast if:

1. $[\![\lceil \star \overset{l}{\Longrightarrow} \star \rceil]\!](v) = v$
2. $[\![\lceil G \overset{l}{\Longrightarrow} \star \rceil]\!](v) = \texttt{Dyn}_G(v)$

3. $[\![\lceil P \xrightarrow{l} \star \rceil]\!](v) = [\![\lceil P \xrightarrow{l} G \rceil]\!](v) \;\gg\!=\; [\![G \xrightarrow{l} \star]\!]$

   if $G \sim P$ and $G \neq P$

4. $[\![\lceil \star \xrightarrow{l} G \rceil]\!](\mathtt{Dyn}_G(u)) = u$

5. $[\![\lceil \star \xrightarrow{l} H \rceil]\!](\mathtt{Dyn}_G(u)) = \Uparrow l$

   if $G \neq H$

6. $[\![\lceil \star \xrightarrow{l} Q \rceil]\!](v) = [\![\lceil \star \xrightarrow{l} H \rceil]\!](v) \;\gg\!=\; [\![\lceil H \xrightarrow{l} Q \rceil]\!]$

   if $H \sim Q$ and $H \neq Q$

7. $[\![\lceil \mathtt{Bool} \xrightarrow{l} \mathtt{Bool} \rceil]\!](v) = v$

8. $[\![\lceil S_1 \to T_1 \xrightarrow{l} S_2 \to T_2 \rceil]\!](\langle \lambda x : S.\ e, E \rangle : c \to d)$

   $= \langle \lambda x : S.\ e, E \rangle : (\lceil S_2 \xrightarrow{l} S_1 \rceil \fatsemi c) \to (d \fatsemi \lceil T_1 \xrightarrow{l} T_2 \rceil)$

9. $[\![\lceil S_1 \times T_1 \xrightarrow{l} S_2 \times T_2 \rceil]\!](\mathtt{cons}\ v_1\ v_2 : c \times d)$

   $= \mathtt{cons}\ v_1\ v_2 : (c \fatsemi \lceil S_1 \xrightarrow{l} S_2 \rceil) \times (d \fatsemi \lceil T_1 \xrightarrow{l} T_2 \rceil)$

10. $[\![\lceil P \xrightarrow{l} Q \rceil]\!](v) = \Uparrow l$

    if $P \not\sim Q$

4.3 The Space-efficient CEK Machine is Correct

In this section, we prove that if C is a Lazy D Cast ADT then

$$eval_{\mathcal{R}(\mathtt{LD})}(e) = o \text{ if and only if } eval_{\mathcal{S}(C)}(e) = o \text{ for all } \emptyset \vdash e : T \text{ and } \vdash o : T$$

and if C is a Lazy UD Cast ADT then

$$eval_{\mathcal{R}(\mathtt{LUD})}(e) = o \text{ if and only if } eval_{\mathcal{S}(C)}(e) = o \text{ for all } \emptyset \vdash e : T \text{ and } \vdash o : T$$

The proof is done via a weak bisimulation.

Fig. 4.3 defines the bisimulation relation between $\mathcal{R}(B)$ and $\mathcal{S}(C)$. It is the smallest congruence relation containing the rules in Fig. 4.3. The relation $\vdash \overline{S \xrightarrow{l} T} \approx c : S \Longrightarrow T$ relates a sequence of cast in $\mathcal{R}(B)$ with one cast in $\mathcal{S}(C)$. This relation is designed to cover all possibly ways to construct

$\boxed{\vdash \overline{S \xrightarrow{l} T} \approx c : S \Longrightarrow T}$

$$\frac{}{\vdash [] \approx id(T) : T \Longrightarrow T} \qquad \frac{}{\vdash [S \xrightarrow{l} T] \approx \lceil S \xrightarrow{l} T \rceil : S \Longrightarrow T}$$

$$\frac{\begin{array}{c}\vdash (T_1 \xrightarrow{l_1} T_2 \ldots T_{i-1} \xrightarrow{l_{i-1}} T_i) \approx c_1 : T_1 \Longrightarrow T_i \\ \vdash (T_i \xrightarrow{l_i} T_{i+1} \ldots T_{n-1} \xrightarrow{l_{n-1}} T_n) \approx c_2 : T_i \Longrightarrow T_n\end{array}}{\vdash (T_1 \xrightarrow{l_1} T_2 \ldots T_{n-1} \xrightarrow{l_{n-1}} T_n) \approx (c_1 \fatsemi c_2) : T_1 \Longrightarrow T_n}$$

$\boxed{\vdash K \approx K : S \Longrightarrow T}$

$$\frac{\vdash K \approx k' : S_n \Longrightarrow T \qquad \vdash (S_1 \xrightarrow{l_1} S_2 \ldots S_{n-1} \xrightarrow{l_{n-1}} S_n) \approx c : S_1 \Longrightarrow S_n}{\vdash [\Box : S_1 \xrightarrow{l_1} S_2] \ldots [\Box : S_{n-1} \xrightarrow{l_{n-1}} S_n] K \approx [\Box : c'] k' : S_1 \Longrightarrow T}$$

$\boxed{\vdash v \approx v : T}$

$$\frac{\begin{array}{c}\Gamma, x : S_1 \vdash e : T_1 \qquad \Gamma \vdash E \approx E' \\ \vdash (S_n \xrightarrow{l_{n-1}} S_{n-1} \ldots S_2 \xrightarrow{l_1} S_1) \approx c : S_n \Longrightarrow S_1 \\ \vdash (T_1 \xrightarrow{l_1} T_2 \ldots T_{n-1} \xrightarrow{l_{n-1}} T_n) \approx d : T_1 \Longrightarrow T_n \\ v = \langle \lambda x : S_1.\ e, E \rangle : S_1 \to T_1 \xrightarrow{l_1} S_2 \to T_2 \ldots : S_{n-1} \to T_{n-1} \xrightarrow{l_{n-1}} S_n \to T_n\end{array}}{\vdash v \approx (\langle \lambda x : S_1.\ e, E' \rangle : c \to d) : S_n \to T_n}$$

$$\frac{\begin{array}{c}\vdash v_1 \approx v_1' : S_1 \qquad \vdash v_2 \approx v_2' : T_1 \\ \vdash (S_1 \xrightarrow{l_1} S_2 \ldots S_{n-1} \xrightarrow{l_{n-1}} S_n) \approx c : S_1 \Longrightarrow S_n \\ \vdash (T_1 \xrightarrow{l_1} T_2 \ldots T_{n-1} \xrightarrow{l_{n-1}} T_n) \approx d : T_1 \Longrightarrow T_n \\ v = \mathtt{cons}\ v_1\ u_1 : S_1 \times T_1 \xrightarrow{l_1} S_2 \times T_2 \ldots : S_{n-1} \times T_{n-1} \xrightarrow{l_{n-1}} S_n \times T_n\end{array}}{\vdash v \approx (\mathtt{cons}\ v_2\ u_2 : c \times d) : S_n \times T_n}$$

$\boxed{\Gamma \vdash E \approx E}$

$$\frac{}{\emptyset \vdash \emptyset \approx \emptyset} \qquad \frac{\Gamma \vdash E \approx E' \qquad v \approx v' : T}{\Gamma, x : T \vdash E[x := v] \approx E'[x := v']}$$

$\boxed{\vdash s \approx s : T}$

$$\frac{}{\vdash \Uparrow l \approx \Uparrow l : T} \qquad \frac{\vdash v \approx v' : T}{\vdash \Downarrow v \approx \Downarrow v' : T} \qquad \frac{\vdash v \approx v' : S \quad \vdash K \approx K' : S \Longrightarrow T}{\vdash \langle v, K \rangle \approx \langle v, K' \rangle : T} \qquad \frac{\Gamma \vdash e : S \quad \Gamma \vdash E \approx E' \quad \vdash K \approx K' : S \Longrightarrow T}{\vdash \langle e, E, K \rangle \approx \langle e, E', K' \rangle : T}$$

Figure 4.3: Bisimulation between $\mathcal{R}(B)$ and $\mathcal{S}(C)$. It is the smallest congruence relation containing the shown rules. Everything to the left (resp. right) of the symbol $\approx$ is in $\mathcal{R}(B)$ (resp. $\mathcal{S}(C)$).

a cast representation. The empty sequence is related to the identity cast. A $\mathcal{R}(B)$ cast is related to its translation. Finally, cast sequencing preserves the bisimulation relation. We remark that this bisimulation relation is unusual in that it uses some ADT operations in its definition, such as $id(T)$, $\lceil S \xRightarrow{l} T \rceil$, and $c \fatsemi c$, which is necessary because $\mathcal{S}(C)$ and the bisimulation is parameterized over the Cast ADT.

The bisimulation relation is mutually defined on continuations, written $\vdash K \approx K : S \Longrightarrow T$ and $\vdash K \approx k : S \Longrightarrow T$, where S and T are their source and target types respectively, values, written $\vdash v \approx v : T$, where T is their type, and environments, written $\Gamma \vdash E \approx E$, where Γ is their type environment. The bisimulation relation for values is designed to make Lemma 1 and Lemma 2 (described below) true, that is, if $\vdash v \approx v' : S$ then

$$\vdash \mathit{applyCast}(v, S \xRightarrow{l} T) \approx [\![\lceil S \xRightarrow{l} T \rceil]\!](v') : T$$

So if two values of type I are related, injecting them to $\star$ gives related values. To relate function closures, the domain and codomain casts should be related. Closures in $\mathcal{R}(B)$ are proxied with a sequence of function casts, which can be viewed as two cast sequences in the obvious way. The rule for products is similar.

One nice aspect of using abstract machines for this proof is that the expressions in the machine are simply related by syntactic equality.

Lemma 1 (Lazy D *applyCast* Preserves Bisimulation)**.** *Assume C is a Lazy D Cast ADT, v is a value in $\mathcal{R}(\texttt{LD})$, v' is a value in $\mathcal{S}(C)$, and $\vdash v \approx v' : S$*

$$\vdash \mathit{applyCast}(v, S \xRightarrow{l} T) \approx [\![\lceil S \xRightarrow{l} T \rceil]\!](v') : T$$

Proof. Case splitting whether S is shallowly consistent with T. If yes, case splitting how they are shallowly consistent. Then apply properties of Lazy D Cast when applicable. □

Lemma 2 (Lazy UD *applyCast* Preserves Bisimulation)**.** *Assume C implements Lazy UD Cast ADT, v is a value in $\mathcal{R}(\texttt{LUD})$, v' is a value in $\mathcal{S}(C)$, and $\vdash v \approx v' : S$*

$$\vdash \mathit{applyCast}(v, S \xRightarrow{l} T) \approx [\![\lceil S \xRightarrow{l} T \rceil]\!](v') : T$$

Proof. Case splitting whether S is shallowly consistent with T. If yes, case splitting how they are shallowly consistent. Then apply properties of Lazy UD Cast when applicable. □

We now come to the main lemma, the weak bisimulation between $\mathcal{R}(B)$ and $\mathcal{S}(C)$.

Lemma 3 (Lazy D Weak Bisimulation between $\mathcal{R}(\texttt{LD})$ and $\mathcal{S}(C)$)**.** *Assume C implements Lazy D Cast ADT and s_1 is a state in $\mathcal{R}(\texttt{LD})$ and s_1' is a state in $\mathcal{S}(C)$ and $\vdash s_1 \approx s_1' : T$ then either*

1. *$s_1 = \Uparrow l$ and $s_1' = \Uparrow l$ for some l, or*
2. *$s_1 = \Downarrow v$ and $s_1' = \Downarrow v'$ and $\vdash v \approx v' : T$ for some v and v', or*
3. *$s_1 \longmapsto^+ s_2$ and $s_1' \longmapsto^+ s_2'$ and $\vdash s_2 \approx s_2' : T$*

Lemma 4 (Lazy UD Weak Bisimulation between $\mathcal{R}(\texttt{LUD})$ and $\mathcal{S}(C)$)**.** *Assume C implements Lazy UD Cast ADT and $\vdash s_1 \approx s_1' : T$ then either*

1. *$s_1 = \Uparrow l$ and $s_1' = \Uparrow l$ for some l, or*
2. *$s_1 = \Downarrow v$ and $s_1' = \Downarrow v'$ and $\vdash v \approx v' : T$ for some v and v', or*
3. *$s_1 \longmapsto^+ s_2$ and $s_1' \longmapsto^+ s_2'$ and $\vdash s_2 \approx s_2' : T$*

Theorem 1 (Correctness of Lazy D $\mathcal{S}(C)$)**.** *Suppose C is a Lazy D Cast ADT*

$$\mathit{eval}_{\mathcal{R}(B)}(e) = o \text{ if and only if } \mathit{eval}_{\mathcal{S}(C)}(e) = o \text{ for all } \emptyset \vdash e : T \text{ and } \vdash o : T$$

Proof. By the definitions of *eval*, we need to show

$$\langle e, \emptyset, \square \rangle \longmapsto^*_{\mathcal{R}(B)} s \text{ if and only if } \langle e, \emptyset, \mathit{mkCont}(\square) \rangle \longmapsto^*_{\mathcal{S}(C)} s'$$

where s and s' are bisimuated final states, i.e. of the form $\Downarrow v$ or $\Uparrow l$. The initial states are related by the bisimulation relation, so we can prove this theorem with Lemma 3. □

Theorem 2 (Correctness of Lazy UD $\mathcal{S}(C)$)**.** *Suppose C is a Lazy UD Cast ADT*

$$\mathit{eval}_{\mathcal{R}(B)}(e) = o \text{ if and only if } \mathit{eval}_{\mathcal{S}(C)}(e) = o \text{ for all } \emptyset \vdash e : T \text{ and } \vdash o : T$$

Proof. Similar to Theorem 1. □

Chapter 5

Coercions in Normal Form as Cast ADT

In this chapter, I review the coercions [Henglein, 1994, Herman et al., 2010] and the normal form of Siek et al. [2015a] (Lazy UD) and Kuhlenschmidt et al. [2019] (Lazy D). I assume a basic familiarity with coercions, and suggest that readers unfamiliar with coercions to familiarize themselves with [Siek and Garcia, 2012] and [Siek et al., 2015a]. At the end of this chapter, I show that Lazy UD coercions in normal form is a Lazy UD Cast ADT, and conjecture that Lazy D coercions in normal form is a Lazy D Cast ADT.

5.1 Background

Fig. 5.1 reviews the grammar for coercions, written c. To review, an injection $I!$ takes a value from an injectable type I to type $\star$. An injectable type is simply a type that can be cast directly to and from $\star$. The definition of injectable types depends on the blame strategy, as in Chapter 2. A projection $I?^l$ takes a value from type $\star$ to type I, or halts the program and blames l if the value is inconsistent with the target type. The coercion $\mathtt{id}_T$ is the identity, $\bot^l$ is the coercion that always fails and blames l, and $(c; d)$ applies c and then d in sequence. The function coercion $c \to d$ applies

Injectable types (Lazy D)	I, J	::=	$T \to T \mid \mathtt{Bool}$
Injectable types (Lazy UD)	I, J	::=	$\star \to \star \mid \mathtt{Bool}$
Coercions	c, d	::=	$\mathtt{id}_T \mid c \to d \mid c \times d \mid \bot^l \mid I! \mid I?^l \mid c; d$
Coercions in normal form	s, t	::=	$\mathtt{id}_\star \mid I?^l; i \mid i$
Final Coercions	i	::=	$\bot^l \mid g; I! \mid g$
Middle Coercions	g	::=	$\mathtt{id}_{\mathtt{Bool}} \mid s \to t \mid s \times t$

Figure 5.1: Syntax of coercions and normal forms à la [Siek and Garcia, 2012].

c to the argument of a function and d to the return value.

Coercions come with a reduction relation so it is natural to ask about their normal forms. Fig. 5.1 also defines the syntax of coercion in normal form, written s or t. Conceptually, a coercion in normal form has three parts, all of which are optional. It may start with a projection $I?^l$, followed by a middle coercion (g) corresponding to some type constructor, and then concluded with an injection $I!$ or a failure $\bot^l$. While this is a simple idea, this is not very obvious in the syntax.

5.2 Coercions in normal form as Cast ADT

Coercions in normal form implement the Cast ADT (Definition 3). The translation and composition functions can be found in [Siek and Garcia, 2012] and [Siek et al., 2015a]. The *id* function is defined in the obvious way.

Theorem 3. *Lazy UD Coercions in Normal Form implement the Lazy UD Cast ADT*

Conjecture 1. *Lazy D Coercions in Normal Form implement the a Lazy D Cast ADT*

5.3 Coercions in Normal Form in Mechanized Metatheory

The composition function of coercions in normal form is not structurally recursive, hence was believed requiring extra termination proof (at least in Agda). Jeremy Siek formalizes Lazy UD coercions in normal form and spends about 300 hundred lines in composition. Deyaaeldeen Almahallawi formalizes Lazy D coercions in normal form and also spends hundred of lines. In contrast, threesomes has a structurally recursive composition and a direct translation should work without any termination proof.

To show that Lazy UD coercions in normal form is a Cast ADT, I also define the composition function in Agda. My formalization, however, spends only 22 lines. My trick is introducing several helper functions, including $g \fatsemi g = g$ and $g \fatsemi i = i$ and $i \fatsemi s = i$. I and Deyaaeldeen have tried to apply the same approach to Lazy D, but fail.

5.4 Coercions in Normal Form in Language Implementation

A straightforward implementation of coercions in normal forms has a memory representation that is not very compact. Grift implements sequencing coercions $c; c$ as a pair, which takes two memory boxes. As a result, coercions like $\texttt{Bool}?^{l}; \texttt{id}_{\texttt{Bool}}; \texttt{Bool}!$ would use two pairs (i.e. four memory boxes). Such representation is not very desirable because coercions about base type might occur frequently in real programs.

Hypercoercions, a new cast representation introduced in Chapter 7 is able to fit the above coercion in 64 bits. Thus it is considered better than coercions in normal form in language implementation.

Chapter 6

Threesomes as Cast ADT

In this chapter I review the Lazy UD threesomes. I prove that Lazy UD threesomes is a Lazy UD Cast ADT and conjecture that Lazy D threesomes is a Lazy D Cast ADT.

Threesomes, introduced by Siek and Wadler [2010] and further studied by Garcia [2013], is based on the observation that any cast is, roughly speaking, equivalent to a downcast (going more precise) followed by an upcast. For example, we might view $(\texttt{Bool} \to \star) \overset{l}{\Longrightarrow} (\star \to \texttt{Bool})$ as $(\texttt{Bool} \to \star) \overset{l}{\Longrightarrow} (\texttt{Bool} \to \texttt{Bool}) \overset{l}{\Longrightarrow} (\star \to \texttt{Bool})$. There are three types involving in a threesome, hence the name [Siek and Wadler, 2010]. And the threesome way of writing this cast is

$$(\texttt{Bool} \to \star) \xRightarrow{\texttt{Bool}^l \to^{\epsilon} \texttt{Bool}^l} (\star \to \texttt{Bool})$$

The type in the middle is the *meet* of source and target. The meet partial function is defined in Chapter 2. Intuitively, the meet contains all information we know about the source and target. Every type constructor in the meet is decorated with an optional label, which describes whether there is a projection into the corresponding ground type.

Fig. 6.1 defines the syntax and typing judgment of Lazy UD threesomes. A threesome is a source type, a labeled middle type, and a target type. A labeled type is either the identity cast for $\star$, a failure and an optional label, or a labeled pre-type and an optional label. The optional label means

Threesomes	c, d	$::=$	$S \overset{\hat{T}}{\Longrightarrow} T$
Labeled Types	$\hat{S}, \hat{T}$	$::=$	$\star \mid \bot^{lG^p} \mid \hat{P}^p$
Labeled Pre-types	$\hat{P}, \hat{Q}$	$::=$	$\texttt{Bool} \mid \hat{S} \to \hat{T} \mid \hat{S} \times \hat{T}$
Optional Labels	p, q	$::=$	$\epsilon \mid l$
Injectable types	I, J	$::=$	$\star \to \star \mid \texttt{Bool}$

Figure 6.1: Syntax of Lazy UD threesomes.

projection when there is a label.

Theorem 4. *Lazy UD Threesomes implement the Lazy UD Cast ADT*

Theorem 5 (Lazy UD Threesomes are Correct)**.** *Let C be the Lazy UD threesomes,*

$$eval_{\mathcal{R}(\texttt{LUD})}(e) = o \text{ if and only if } eval_{\mathcal{S}(C)}(e) = o \text{ for all } \emptyset \vdash e : T \text{ and } \vdash o : T$$

Conjecture 2. *Lazy D Threesomes implement the Lazy D Cast ADT*

Conjecture 3 (Lazy D Threesomes are Correct)**.** *Let C be the Lazy D threesomes,*

$$eval_{\mathcal{R}(\texttt{LD})}(e) = o \text{ if and only if } eval_{\mathcal{S}(C)}(e) = o \text{ for all } \emptyset \vdash e : T \text{ and } \vdash o : T$$

Chapter 7

Hypercoercions, a New Cast Representation

In this chapter I describe a new cast representation called hypercoercions. It comes with two flavors: Lazy D hypercoercions and Lazy UD one. The design of hypercoercions is motivated by the observation that a normal coercion has at most three parts. Hypercoercions make this structure explicit: a hypercoercion c is either $\mathtt{id}_\star$, the identity cast for $\star$, or it contains three parts: a head, middle, and tail, as defined in Fig. 7.1.

- a head is either a projection or the no-op;
- a middle m involves coercions that preserve a type constructor, i.e., coercions between Boolean, function, and pair, and failure coercions;
- a tail is either an injection or the no-op.

Section 7.1 implements the Cast ADT with the Lazy D hypercoercions and proves that Lazy D hypercoercions is a Lazy D Cast ADT. Section 7.2 does the Lazy UD counterparts. Finally, I argue in Section 7.3 that hypercoercions would consume less space than coercions in normal form and hence are more suitable for implementation.

7.1 Lazy D Hypercoercions

Fig. 7.2 and Fig. 7.3 defines the functions $c \fatsemi c$ that compose Lazy D hypercoercions. We use these functions to implement the Cast ADT defined in Chapter 4.

The $c \fatsemi c$ operator is the key to compressing coercions and maintaining space-efficiency. The composition operator for Coercions in Normal Form requires the target type of the first coercion to match the source type of the second. This requirement, however, imposes a considerable difficulty in termination proof of the composition of Lazy D Coercions in Normal Form, reported by Kuang-Chen Lu and Deyaaeldeen Almahallawi. So it is useful to relax this restriction for the Lazy D tracking

Syntax of Hypercoercions

$$\begin{array}{lrcl} \text{Hypercoercions} & c & ::= & \mathtt{id}_\star \mid (h; m; t) \\ \text{Heads} & h & ::= & \epsilon \mid I?^l \\ \text{Middles} & m & ::= & \mathtt{Bool} \mid c \to c \mid c \times c \mid \bot^l \\ \text{Tails} & t & ::= & \epsilon \mid I! \end{array}$$

$\boxed{id(T) = c}$

$$\begin{array}{rcl} id(\star) & = & \mathtt{id}_\star \\ id(P) & = & (\epsilon; id(P); \epsilon) \end{array}$$

$\boxed{id_m(P) = m}$

$$\begin{array}{rcl} id(\mathtt{Bool}) & = & \mathtt{Bool} \\ id(S \to T) & = & id(S) \to id(T) \\ id(S \times T) & = & id(S) \times id(T) \end{array}$$

$\boxed{\vdash c : S \Longrightarrow T}$ Hypercoercion typing

$$\frac{}{\vdash \mathtt{id}_\star : \star \Longrightarrow \star} \qquad \frac{\vdash h : S \Longrightarrow P \quad \vdash m : P \Longrightarrow Q \quad \vdash t : Q \Longrightarrow T}{\vdash (h; m; t) : S \Longrightarrow T}$$

$\boxed{\vdash h : T \Longrightarrow P}$ Head typing

$$\frac{}{\vdash \epsilon : P \Longrightarrow P} \qquad \frac{}{\vdash I?^l : \star \Longrightarrow I}$$

$\boxed{\vdash m : T \Longrightarrow T}$ Middle typing

$$\frac{}{\vdash \bot^l : P \Longrightarrow Q} \qquad \frac{}{\vdash \mathtt{Bool} : \mathtt{Bool} \Longrightarrow \mathtt{Bool}}$$

$$\frac{\vdash c : S_2 \Longrightarrow S_1 \quad \vdash d : T_1 \Longrightarrow T_2}{\vdash c \to d : S_1 \to T_1 \Longrightarrow S_2 \to T_2} \qquad \frac{\vdash c : S_1 \Longrightarrow S_2 \quad \vdash d : T_1 \Longrightarrow T_2}{\vdash c \times d : S_1 \times T_1 \Longrightarrow S_2 \times T_2}$$

$\boxed{\vdash t : P \Longrightarrow T}$ Tail typing

$$\frac{}{\vdash \epsilon : P \Longrightarrow P} \qquad \frac{}{\vdash I! : I \Longrightarrow \star}$$

Figure 7.1: Definition of hypercoercions

$$\text{Gaps} \quad g \quad ::= \quad \epsilon \mid S \stackrel{l}{\Longrightarrow} T$$

$\boxed{c \fatsemi c = c}$ Composition of Lazy D hypercoercions

$$c_1 \fatsemi c_2 \quad = \quad c_1 \fatsemi \epsilon \fatsemi c_2$$

$\boxed{c \fatsemi g \fatsemi c = c}$

$$\begin{array}{rcl}
\mathtt{id}_\star \fatsemi g \fatsemi \mathtt{id}_\star & = & \mathtt{id}_\star \\
\mathtt{id}_\star \fatsemi g \fatsemi (h; m; t) & = & (\mathit{mkProj}(g, h); m; t) \\
(h; m; t) \fatsemi g \fatsemi \mathtt{id}_\star & = & (h; m; \mathit{mkInj}(t, g)) \\
(h_1; m_1; t_1) \fatsemi g \fatsemi (h_2; m_2; t_2) & = & (h_1; (m_1 \fatsemi \mathit{mkGap}(t_1, g, h_2) \fatsemi m_2); t_2)
\end{array}$$

$\boxed{\mathit{mkProj}(g, h)}$

$$\begin{array}{rcl}
\mathit{mkProj}(g, P?^l) & = & P?^l \\
\mathit{mkProj}(\star \stackrel{l}{\Longrightarrow} P, \epsilon) & = & P?^l
\end{array}$$

$\boxed{\mathit{mkInj}(t, g)}$

$$\begin{array}{rcl}
\mathit{mkInj}(P!, g) & = & P! \\
\mathit{mkInj}(\epsilon, P \stackrel{l}{\Longrightarrow} \star) & = & P!
\end{array}$$

$\boxed{\mathit{mkGap}(t, g, h)}$

$$\begin{array}{rcl}
\mathit{mkGap}(\epsilon, g, \epsilon) & = & g \\
\mathit{mkGap}(\epsilon, P \stackrel{l'}{\Longrightarrow} \star, Q?^l) & = & P \stackrel{l}{\Longrightarrow} Q \\
\mathit{mkGap}(P!, g, Q?^l) & = & P \stackrel{l}{\Longrightarrow} Q \\
\mathit{mkGap}(P!, \star \stackrel{l}{\Longrightarrow} Q, \epsilon) & = & P \stackrel{l}{\Longrightarrow} Q
\end{array}$$

Figure 7.2: Composition of Lazy D Hypercoercions

$\boxed{m \fatsemi g \fatsemi m = m}$

$$
\begin{array}{rcll}
\bot^l \fatsemi g \fatsemi m_2 & = & \bot^l & \\
m_1 \fatsemi P \stackrel{l}{\Longrightarrow} Q \fatsemi m_2 & = & \bot^l & \text{if } P \not\smile Q \text{ and } \forall l.m_1 \neq \bot^l \\
m_1 \fatsemi g \fatsemi \bot^l & = & \bot^l & \text{if } good(g) \text{ and } \forall l.m_1 \neq \bot^l \\
\texttt{Bool} \fatsemi g \fatsemi \texttt{Bool} & = & \texttt{Bool} & \\
(c_1 \to d_1) \fatsemi g \fatsemi (c_2 \to d_2) & = & (c_2 \fatsemi dom(g) \fatsemi c_1) \to (d_1 \fatsemi cod(g) \fatsemi d_2) & \\
(c_1 \times d_1) \fatsemi g \fatsemi (c_2 \times d_2) & = & (c_1 \fatsemi car(g) \fatsemi c_2) \times (d_1 \fatsemi cdr(g) \fatsemi d_2) &
\end{array}
$$

$\boxed{good(g)}$

$$\frac{}{good(\epsilon)} \qquad \frac{P \smile Q}{good(P \stackrel{l}{\Longrightarrow} Q)}$$

$\boxed{dom(g) = g}$

$$
\begin{array}{rcl}
dom(\epsilon) & = & \epsilon \\
dom(S_1 \to T_1 \stackrel{l}{\Longrightarrow} S_2 \to T_2) & = & S_2 \stackrel{l}{\Longrightarrow} S_1
\end{array}
$$

$\boxed{cod(g) = g}$

$$
\begin{array}{rcl}
cod(\epsilon) & = & \epsilon \\
cod(S_1 \to T_1 \stackrel{l}{\Longrightarrow} S_2 \to T_2) & = & T_1 \stackrel{l}{\Longrightarrow} T_2
\end{array}
$$

$\boxed{car(g) = g}$

$$
\begin{array}{rcl}
car(\epsilon) & = & \epsilon \\
car(S_1 \times T_1 \stackrel{l}{\Longrightarrow} S_2 \times T_2) & = & S_1 \stackrel{l}{\Longrightarrow} S_2
\end{array}
$$

$\boxed{cdr(g) = g}$

$$
\begin{array}{rcl}
cdr(\epsilon) & = & \epsilon \\
cdr(S_1 \times T_1 \stackrel{l}{\Longrightarrow} S_2 \times T_2) & = & T_1 \stackrel{l}{\Longrightarrow} T_2
\end{array}
$$

Figure 7.3: Composition of Lazy D Hypercoercions (Cont.)

strategy. We shall need optional source cast, written g, that range over ϵ or $S \stackrel{l}{\Longrightarrow} T$, to maintain the gap between the hypercoercions to be composed. The source cast is mandatory when the target type of the first hypercoercion does not match the source type of the second. Then we write the composition of hypercoercions as $c_1 \fatsemi g \fatsemi c_2$. When both c_1 and c_2 are $\mathtt{id}_\star$, their composition is also $\mathtt{id}_\star$. When the first hypercoercion is the $\mathtt{id}_\star$, the output must start with a projection, in which case we invoke the helper function *mkProj*. In *mkProj*, if the head of c_2 is a projection, the result is c_2. Otherwise, we need a label to build the projection. Since the head of c_2 is the no-op, its source type must be a pretype. Thus we know g must be a source cast and we use it to construct the projection. Going back to $c \fatsemi g \fatsemi c$, when the second hypercoercion is the $\mathtt{id}_\star$, the output must end with an injection, in which case we invoke the helper function *mkInj*. Finally, when both inputs are of the form $(h; m; t)$, we compose the middle with an updated gap. The second and third cases of $mkGap(t, g, h)$ is interesting, where we choose the label in projection to be responsible for potential inconsistency.

Fig. 7.3 defines the middle composition $m \fatsemi g \fatsemi m$. When the first middle is a failure ($\bot^l$), the result is the second middle. In all the remaining cases, the first middle is not a failure. When the gap is a cast between shallowly inconsistent types (i.e. having different top-most constructors). the output is a failure. When the gap is good but the second middle is a failure, the output is the same failure. A gap g is good if and only if there is not P and l and Q such that $g = P \stackrel{l}{\Longrightarrow} Q$ and $P \not\sim Q$. The remaining cases are straightforward recursion.

Agda is able to understand that $c \fatsemi g \fatsemi c$ is terminating. The reason might be that in $c \fatsemi g \fatsemi c$ both casts become smaller in recursive calls, (although their position is swapped in the function case).

The definition of translation $\lceil S \stackrel{l}{\Longrightarrow} T \rceil$ is straightforward (Fig. 7.4).

Fig. 7.5 defines the cast application function for Lazy D hypercoercions. Applying the identity cast for the dynamic type succeeds immediately. Otherwise, we apply head, middle, and tail in sequence. In projection case of head application, a new middle is constructed and applied to the

$\boxed{\lceil S \overset{l}{\Longrightarrow} T \rceil = c}$ Translation of Lazy D hypercoercions

$$\lceil S \overset{l}{\Longrightarrow} T \rceil \quad = \quad id(S) \fatsemi (S \overset{l}{\Longrightarrow} T) \fatsemi id(T)$$

Figure 7.4: Translatin of Lazy D Hypercoercions

$\boxed{[\![c]\!](v) = r}$

$$\begin{aligned}
[\![\mathtt{id}_\star]\!](v) &= v \\
[\![(h; m; t)]\!](v) &= ([\![h]\!]_h(v) \gg\!= [\![m]\!]_m) \gg\!= [\![t]\!]_t
\end{aligned}$$

$\boxed{[\![h]\!]_h(v) = r}$

$$\begin{aligned}
[\![\epsilon]\!](v) &= v \\
[\![Q?^l]\!](\mathtt{Dyn}_P(u)) &= [\![id_m(P) \fatsemi P \overset{l}{\Longrightarrow} Q \fatsemi id_m(Q)]\!]_m(u)
\end{aligned}$$

$\boxed{[\![m]\!]_m(v) = r}$

$$\begin{aligned}
[\![\bot^l]\!](v) &= \Uparrow l \\
[\![\mathtt{Bool}]\!](v) &= v \\
[\![d_1 \to d_2]\!](\langle \lambda x : S.\ e, E \rangle : c_1 \to c_2) &= \langle \lambda x : S.\ e, E \rangle : (d_1 \fatsemi c_1) \to (c_2 \fatsemi d_2) \\
[\![d_1 \times d_2]\!](\mathtt{cons}\ v_1\ v_2 : c_1 \times c_2) &= \mathtt{cons}\ v_1\ v_2 : (c_1 \fatsemi d_1) \times (c_2 \fatsemi d_2)
\end{aligned}$$

$\boxed{[\![t]\!]_t(v) = r}$

$$\begin{aligned}
[\![\epsilon]\!](v) &= v \\
[\![P!]\!](v) &= \mathtt{Dyn}_P(v)
\end{aligned}$$

Figure 7.5: Application of Lazy D Hypercoercions

injected value. Middle applications raises an error if the middle is a failure and extend value proxies otherwise.

Proposition 4. *Lazy D Hypercoercions implement the Lazy D Cast ADT*

Theorem 6 (Lazy D Hypercoercions are Correct)**.** *Let H be the Lazy D hypercoercions,*

$$\mathit{eval}_{\mathcal{R}(\mathtt{LD})}(e) = o \text{ if and only if } \mathit{eval}_{\mathcal{S}(H)}(e) = o \text{ for all } \emptyset \vdash e : T \text{ and } \vdash o : T$$

7.2 Lazy UD Hypercoercions

Hypercoercions for the Lazy UD blame tracking strategy have the same syntax as for Lazy D (Fig. 7.1), but the definitions of $id(T)$ and $\lceil S \stackrel{l}{\Longrightarrow} T \rceil$ and $c \fatsemi c$, differ from Lazy D. Here composition $c_1 \fatsemi c_2$ makes the usual assumption that the target type of c_1 matches the source type of c_2. The definition of composition for Lazy UD is particularly straightforward. The first two lines say that $\mathtt{id}_\star$ acts as the identity on both the left and right. The third line handles the case when both hypercoercions have three parts. The first three lines of the middle composition $m \fatsemi t \fatsemi h \fatsemi m$ function match its arguments in sequential order to determine which label to blame. The forth and last line is straightforward recursion.

The translation function $\lceil S \stackrel{l}{\Longrightarrow} T \rceil$ is defined in Fig. 7.7. Prior presentations of this function do not use auxiliary functions, as we do here. The reason that we introduce the auxiliary functions *castToDyn* and *castFromDyn*, is to ensure that each of them is structurally recursive, which makes them straightforward to define in Agda.

Fig. 7.8 defines the cast application function for Lazy UD hypercoercions. Applying the identity cast for the dynamic type succeeds immediately. Otherwise, we apply head, middle, and tail in sequence. In the two projection cases of head application, the result depends on whether the source ground type (G) is the same as the target (H). Middle applications raises an error if the middle is a failure and extend value proxies otherwise.

$\boxed{c \fatsemi c = c}$

$$
\begin{array}{rcll}
\mathtt{id}_\star & \fatsemi & d & = & d \\
(h; m; t) & \fatsemi & \mathtt{id}_\star & = & (h; m; t) \\
(h_1; m_1; t_1) & \fatsemi & (h_2; m_2; t_2) & = & (h_1; (m_1 \fatsemi t_1 \fatsemi h_2 \fatsemi m_2); t_2)
\end{array}
$$

$\boxed{m \fatsemi t \fatsemi h \fatsemi m = m}$

$$
\begin{array}{rcrcrcll}
\bot^l & \fatsemi & t & \fatsemi & h & \fatsemi & m_2 & = & \bot^l & \\
m_1 & \fatsemi & G! & \fatsemi & H?^l & \fatsemi & m_2 & = & \bot^l & \text{if } G \neq H \text{ and } \forall l. m_1 \neq \bot^l \\
m_1 & \fatsemi & t & \fatsemi & h & \fatsemi & \bot^l & = & \bot^l & \text{if } t \ R \ h \text{ and } \forall l. m_1 \neq \bot^l \\
m_1 & \fatsemi & t & \fatsemi & h & \fatsemi & m_2 & = & m_1 \fatsemi m_2 & \text{if } \forall l. m_2 \neq \bot^l \text{ and } t \ R \ h \text{ and } \forall l. m_1 \neq \bot^l
\end{array}
$$

$\boxed{m \fatsemi m = m}$

$$
\begin{array}{rcrcl}
\mathtt{Bool} & \fatsemi & \mathtt{Bool} & = & \mathtt{Bool} \\
c_1 \to d_1 & \fatsemi & c_2 \to d_2 & = & c_2 \fatsemi c_1 \to d_1 \fatsemi d_2 \\
c_1 \times d_1 & \fatsemi & c_2 \times d_2 & = & c_2 \fatsemi c_1 \times d_1 \fatsemi d_2
\end{array}
$$

$\boxed{t \ R \ h}$

$$
\frac{}{\epsilon \ R \ \epsilon} \qquad \frac{}{G! \ R \ G?^l}
$$

Figure 7.6: Lazy UD Hypercoercions (1): composition

$\boxed{\lceil S \overset{l}{\Longrightarrow} T \rceil = c}$

$$
\begin{array}{rcll}
\lceil \star \overset{l}{\Longrightarrow} \star \rceil & = & \mathtt{id}_\star & \\
\lceil \star \overset{l}{\Longrightarrow} Q \rceil & = & \mathit{castFromDyn}(l, Q) & \\
\lceil P \overset{l}{\Longrightarrow} \star \rceil & = & \mathit{castToDyn}(P, l) & \\
\lceil P \overset{l}{\Longrightarrow} Q \rceil & = & (\epsilon; \bot^l; \epsilon) & \text{if } P \not\sim Q \\
\lceil \mathtt{Bool} \overset{l}{\Longrightarrow} \mathtt{Bool} \rceil & = & (\epsilon; \mathtt{Bool}; \epsilon) & \\
\lceil S_1 \to T_1 \overset{l}{\Longrightarrow} S_2 \to T_2 \rceil & = & (\epsilon; \lceil S_2 \overset{l}{\Longrightarrow} S_1 \rceil \to \lceil T_1 \overset{l}{\Longrightarrow} T_2 \rceil; \epsilon) & \\
\lceil S_1 \times T_1 \overset{l}{\Longrightarrow} S_2 \times T_2 \rceil & = & (\epsilon; \lceil S_1 \overset{l}{\Longrightarrow} S_2 \rceil \times \lceil T_1 \overset{l}{\Longrightarrow} T_2 \rceil; \epsilon) &
\end{array}
$$

$\boxed{\mathit{castFromDyn}(Q, l)}$

$$
\begin{array}{rcl}
\mathit{castFromDyn}(\star, l) & = & \mathtt{id}_\star \\
\mathit{castFromDyn}(\mathtt{Bool}, l) & = & (\mathtt{Bool}?^l; \mathtt{Bool}; \epsilon) \\
\mathit{castFromDyn}(S \to T, l) & = & (\star \to \star?^l; \mathit{castToDyn}(S, l) \to \mathit{castFromDyn}(T, l); \epsilon) \\
\mathit{castFromDyn}(S \times T, l) & = & (\star \times \star?^l; \mathit{castFromDyn}(S, l) \times \mathit{castFromDyn}(T, l); \epsilon)
\end{array}
$$

$\boxed{\mathit{castToDyn}(Q, l)}$

$$
\begin{array}{rcl}
\mathit{castToDyn}(\star, l) & = & \mathtt{id}_\star \\
\mathit{castToDyn}(\mathtt{Bool}, l) & = & (\epsilon; \mathtt{Bool}; \mathtt{Bool}!) \\
\mathit{castToDyn}(S \to T, l) & = & (\epsilon; \mathit{castFromDyn}(S, l) \to \mathit{castToDyn}(T, l); \star \to \star!) \\
\mathit{castToDyn}(S \times T, l) & = & (\epsilon; \mathit{castToDyn}(S, l) \times \mathit{castToDyn}(T, l); \star \times \star!)
\end{array}
$$

Figure 7.7: Lazy UD Hypercoercions (2): translation

$\boxed{[\![c]\!]_c(v) = r}$

$$
\begin{array}{rcl}
[\![\mathtt{id}_\star]\!](v) & = & v \\
[\![(h; m; t)]\!](v) & = & ([\![h]\!](v) \gg\!= [\![m]\!]) \gg\!= [\![t]\!]
\end{array}
$$

$\boxed{[\![h]\!]_h(v) = r}$

$$
\begin{array}{rcl}
[\![\epsilon]\!](v) & = & v \\
[\![G?^l]\!](\mathtt{Dyn}_G(u)) & = & u \\
[\![H?^l]\!](\mathtt{Dyn}_G(u)) & = & \Uparrow l \quad \text{if } G \neq H
\end{array}
$$

$\boxed{[\![m]\!]_m(v) = r}$

$$
\begin{array}{rcl}
[\![\bot^l]\!](v) & = & \Uparrow l \\
[\![\mathtt{Bool}]\!](v) & = & v \\
[\![d_1 \to d_2]\!](\langle \lambda x : S.\ e, E \rangle : c_1 \to c_2) & = & \langle \lambda x : S.\ e, E \rangle : (d_1 \fatsemi c_1) \to (c_2 \fatsemi d_2) \\
[\![d_1 \times d_2]\!](\mathtt{cons}\ v_1\ v_2 : c_1 \times c_2) & = & \mathtt{cons}\ v_1\ v_2 : (c_1 \fatsemi d_1) \times (c_2 \fatsemi d_2)
\end{array}
$$

$\boxed{[\![t]\!]_t(v) = r}$

$$
\begin{array}{rcl}
[\![\epsilon]\!](v) & = & v \\
[\![G!]\!](v) & = & \mathtt{Dyn}_G(v)
\end{array}
$$

Figure 7.8: Application of Lazy UD Hypercoercions

Proposition 5. *Lazy UD Hypercoercions implement the Lazy UD Cast ADT*

Theorem 7 (Lazy UD Hypercoercions are Correct)**.** *Let H be the Lazy UD hypercoercions,*

$$eval_{\mathcal{R}(\texttt{LD})}(e) = o \text{ if and only if } eval_{\mathcal{S}(H)}(e) = o \text{ for all } \emptyset \vdash e : T \text{ and } \vdash o : T$$

7.3 Compact Representation of Hypercoercions

Hypercoercions enable a bit-level representation that is particularly compact for identity coercions and coercions that project from base types or inject to base types. (Here `Bool` is the only base type, but in a real language the base types would include integers, Booleans, etc.) We conjecture that such coercions occur more frequently than the more complex coercions (e.g. between function types), especially because such coercions appear in the leaves of complex coercions.

Furthermore, values of base types are often stored in CPU registers, so it would be nice for coercions on base types to also fit in registers, that is, in 64 bits, so that applying a coercion to a value of base type would not require access to main memory, which is an order of magnitude slower than accessing registers on a modern CPU. Coercions involving non-base types, such as function types, may be arbitrarily deep, so in those cases, the hypercoercion representation has to be a pointer to a heap-allocated structure.

Here is a sketch for the bit-level representation for hypercoercions.

- 1 bit to differentiate between $\texttt{id}_\star$ and three-part coercions.
- 1 bit to differentiate between middle coercions of base type versus non-base type.
- If the middle is a non-base type or bottom ($\bot^l$), then 61 bits represent a pointer to a heap-allocated structure. Heap allocated structures are usually 8-byte aligned, so there are 3 unused bits in a pointer.
- If the middle is of base types, then the remaining 62 bits are used to represent the head (51 bits), middle (10 bit), and tail (1 bit).

- The head requires 1 bit to differentiate between ϵ and $P?^l$ and then 50 bits could be used for the label l.
- The middle would use 10 bits to differentiate all the base types.
- The tail requires 1 bits to differentiate between ϵ and $P!$, where P must be equal to the middle type.

The bits for blame labels represent an index into a table of blame information. In the event that a program requires too many blame labels, then the implementation can fallback to using the heap-allocated structure for more coercions.

Chapter 8

Conclusion

Naive implementations of gradual typing have a space leaking problem. Space-efficient cast representations (e.g. coercions in normal form and threesomes) solve this problem by compressing cast information. Using cast representations to achieve space-efficiency is an optimization, hence should preserve the behavior of language implementations. These cast representations have different (albeit similar) correctness proof, which unfortunately hinders the interface between cast representations and the remaining part of the dynamic semantics.

I present Cast ADT, an abstract data type that captures the interface of cast representations, $\mathcal{S}(C)$, an abstract machine parameterized over Cast ADT, and a framework for proving correctness of Cast ADT. The framework states that if C is a Lazy D Cast ADT, $\mathcal{S}(C)$ is equivalent to $\mathcal{R}(\texttt{LD})$ and that if C is a Lazy UD Cast ADT, $\mathcal{S}(C)$ is equivalent to $\mathcal{R}(\texttt{LUD})$İ show with the framework that Lazy UD coercions in normal form and Lazy UD threesomes are correct, and that other known cast representations also implement an appropriate Cast ADT and thus can be proved correct with my framework. I also introduce hypercoercions, a cast representation that is good for both mechanized metatheory and good for implementation, and prove its correctness with the framework.

Bibliography

Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

Matteo Cimini and Jeremy G Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. *ACM SIGPLAN Notices*, 51(1):443–455, 2016.

Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. July 2007.

Matthias Felleisen and Daniel P Friedman. *Control Operators, the SECD-machine, and the [1]-calculus*. Indiana University, Computer Science Department, 1986.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, 2002.

Ronald Garcia. Calculating threesomes, with blame. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 417–428, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500603. URL `http://doi.acm.org/10.1145/2500365.2500603`.

Ronald Garcia, Alison M Clark, and Éric Tanter. Abstracting gradual typing. *ACM SIGPLAN Notices*, 51(1):429–442, 2016.

Álvaro García-Pérez, Pablo Nogueira, and Ilya Sergey. Deriving interpretations of the gradually-typed lambda calculus. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 157–168, 2014.

Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN con-*

ference on Object oriented programming systems languages and applications, pages 231–245, New York, NY, USA, 2005. ACM Press.

Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22 (3):197–230, 1994.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167, 2010.

Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):6, 2010.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. An efficient compiler for the gradually typed lambda calculus. In *Scheme and Functional Programming Workshop*, volume 18, 2018.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G Siek. Toward efficient gradual typing for structural types via coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 517–532, 2019.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007.

Max S New, Daniel R Licata, and Amal Ahmed. Gradual type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL):15, 2019.

Ulf Norell. Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer, 2008.

Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, pages 17–31. Springer, 2009.

Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: Together again for the first time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 425–435, New York, NY, USA, 2015a. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737968. URL http://doi.acm.org/10.1145/2737924.2737968.

Jeremy G Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, pages 68–80. ACM, 2012.

Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 365–376, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706342. URL http://doi.acm.org/10.1145/1706299.1706342.

Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015b.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 456–468, 2016.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 964–974, 2006.

Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic languages*, pages 45–56, 2014.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, pages 1–16. Springer, 2009.

Appendix A

Appendix for Chapter 2

A.1 Types

```
module Types where

open import Relation.Nullary using (Dec; yes; no; ¬_)

open import Data.Empty using (⊥-elim)

open import Relation.Binary.PropositionalEquality using (_≡_; refl; sym; cong)

open import Data.Product using (∃-syntax; _,_; proj₁)

open import Data.Vec using (Vec; replicate; []; _∷_; map)

open import Data.Nat using (ℕ)


infix  99 `_

infix 101 _·_

infix 100 _⇒_

infix 100 _⊗_


data TypeOp : Set where

  `B : TypeOp

  `⊗ : TypeOp

  `⇒ : TypeOp


_=?op_ : (o1 o2 : TypeOp) → Dec (o1 ≡ o2)

`⊗ =?op `⊗ = yes refl

`⊗ =?op `⇒ = no (λ ())

`⇒ =?op `⊗ = no (λ ())

`⇒ =?op `⇒ = yes refl
```

```
‘B =?op ‘B = yes refl
‘B =?op ‘⊗ = no (λ ())
‘B =?op ‘⇒ = no (λ ())
‘⊗ =?op ‘B = no (λ ())
‘⇒ =?op ‘B = no (λ ())

arity : TypeOp → ℕ
arity ‘B = 0
arity ‘⊗ = 2
arity ‘⇒ = 2

record PreType : Set
data Type : Set

record PreType where
  inductive
  constructor _·_
  field
    op : TypeOp
    T* : Vec Type (arity op)

data Type where
  *  : Type
  ‘_ : (P : PreType) → Type

data Polarity : Set where
  - : Polarity
  + : Polarity
```

```
polarity : (op : TypeOp) → Vec Polarity (arity op)
polarity ‘B = []
polarity ‘⊗ = + :: + :: []
polarity ‘⇒ = - :: + :: []

choose : {A : Set} → Polarity → A → A → A
choose - a b = a
choose + a b = b

pattern B       = ‘B · []
pattern _⇒_ S T = ‘⇒ · (S :: T :: [])
pattern _⊗_ S T = ‘⊗ · (S :: T :: [])

data Same : PreType → Set where
  same : ∀ P → Same P

data Ground : PreType → Set where
  ‘B : Ground (B)
  ‘⇒ : Ground (∗ ⇒ ∗)
  ‘⊗ : Ground (∗ ⊗ ∗)

GroundType : Set
GroundType = ∃[ P ] Ground P

GB : GroundType
GB = B , ‘B
```

```
G⇒ : GroundType
G⇒ = * ⇒ * , `⇒

G⊗ : GroundType
G⊗ = * ⊗ * , `⊗

GroundType→PreType : GroundType → PreType
GroundType→PreType = proj₁

ground-unique : ∀ {P} → (G H : Ground P) → G ≡ H
ground-unique `B `B = refl
ground-unique `⇒ `⇒ = refl
ground-unique `⊗ `⊗ = refl

op→ground : TypeOp → PreType
op→ground op = op · replicate *

op→ground-Ground : ∀ op → Ground (op→ground op)
op→ground-Ground `B = `B
op→ground-Ground `⊗ = `⊗
op→ground-Ground `⇒ = `⇒

inv-ground : ∀ {P} → Ground P → ∃[ o ](P ≡ o · (replicate *))
inv-ground `B = `B , refl
inv-ground `⇒ = `⇒ , refl
inv-ground `⊗ = `⊗ , refl

unground : ∀ {P} → Ground P → PreType
```

```
unground {P} gP = P


unground' : ∀ {P} → Ground P → PreType
unground' `B = B
unground' `⇒ = * ⇒ *
unground' `⊗ = * ⊗ *


lem-unground : ∀ {P} → (gP : Ground P) → unground gP ≡ unground' gP
lem-unground `B = refl
lem-unground `⇒ = refl
lem-unground `⊗ = refl


ground? : (P : PreType) → Dec (Ground P)
ground? B = yes `B
ground? (* ⇒ *) = yes `⇒
ground? (* ⇒ (` P)) = no λ ()
ground? ((` P) ⇒ T₂) = no (λ ())
ground? (* ⊗ *) = yes `⊗
ground? (* ⊗ (` P)) = no (λ ())
ground? ((` P) ⊗ T) = no (λ ())


_=?_ : (T1 T2 : Type) → Dec (T1 ≡ T2)
* =? * = yes refl
* =? (` P) = no (λ ())
(` P) =? * = no (λ ())
(` B) =? (` B) = yes refl
(` B) =? (` (T₁ ⇒ T₂)) = no (λ ())
(` B) =? (` (T₁ ⊗ T₂)) = no (λ ())
```

```
(‘ (T₁ ⇒ T₂)) =? (‘ B) = no (λ ())

(‘ (T₁ ⇒ T₂)) =? (‘ (T₃ ⇒ T₄)) with T₁ =? T₃ | T₂ =? T₄

((‘ (T₁ ⇒ T₂)) =? (‘ (.T₁ ⇒ .T₂))) | yes refl | yes refl = yes refl

((‘ (T₁ ⇒ T₂)) =? (‘ (.T₁ ⇒ T₄))) | yes refl | no ¬p = no λ { refl → ¬p refl }

((‘ (T₁ ⇒ T₂)) =? (‘ (T₃ ⇒ T₄))) | no ¬p | p2 = no λ { refl → ¬p refl }

(‘ (T₁ ⇒ T₂)) =? (‘ (T₃ ⊗ T₄)) = no (λ ())

(‘ (T₁ ⊗ T₂)) =? (‘ B) = no (λ ())

(‘ (T₁ ⊗ T₂)) =? (‘ (T₃ ⇒ T₄)) = no (λ ())

(‘ (T₁ ⊗ T₂)) =? (‘ (T₃ ⊗ T₄)) with T₁ =? T₃ | T₂ =? T₄

((‘ (T₁ ⊗ T₂)) =? (‘ (.T₁ ⊗ .T₂))) | yes refl | yes refl = yes refl

((‘ (T₁ ⊗ T₂)) =? (‘ (.T₁ ⊗ T₄))) | yes refl | no ¬p = no λ { refl → ¬p refl }

((‘ (T₁ ⊗ T₂)) =? (‘ (T₃ ⊗ T₄))) | no ¬p | p2 = no λ { refl → ¬p refl }


-- consistency


data _~_ : (T1 T2 : Type) → Set where

  *~* : * ~ *

  *~P : ∀ P → * ~ (‘ P)

  P~* : ∀ P → (‘ P) ~ *

  ~B : (‘ B) ~ (‘ B)

  ~⇒ : ∀ {T1 T2 T3 T4}

    → T1 ~ T3

    → T2 ~ T4

    → (‘ T1 ⇒ T2) ~ (‘ T3 ⇒ T4)

  ~⊗ : ∀ {T1 T2 T3 T4}

    → T1 ~ T3

    → T2 ~ T4

    → (‘ T1 ⊗ T2) ~ (‘ T3 ⊗ T4)
```

```
*~T : ∀ {T} → * ~ T
*~T {*}   = *~*
*~T {‘ P} = *~P P

T~* : ∀ {T} → T ~ *
T~* {*}   = *~*
T~* {‘ P} = P~* P

~refl : ∀ T → T ~ T
~refl * = *~*
~refl (‘ B) = ~B
~refl (‘ (T₁ ⇒ T₂)) = ~⇒ (~refl T₁) (~refl T₂)
~refl (‘ (T₁ ⊗ T₂)) = ~⊗ (~refl T₁) (~refl T₂)

-- shallow consistency

data _⌣_ : (T1 T2 : Type) → Set where
  *⌣* : * ⌣ *
  *⌣P : ∀ P → * ⌣ (‘ P)
  P⌣* : ∀ P → (‘ P) ⌣ *
  ⌣B : (‘ B) ⌣ (‘ B)
  ⌣⇒ : ∀ {T1 T2 T3 T4} → (‘ T1 ⇒ T2) ⌣ (‘ T3 ⇒ T4)
  ⌣⊗ : ∀ {T1 T2 T3 T4} → (‘ T1 ⊗ T2) ⌣ (‘ T3 ⊗ T4)

_⌣?_ : ∀ T1 T2 → Dec (T1 ⌣ T2)
* ⌣? * = yes *⌣*
* ⌣? (‘ P) = yes (*⌣P P)
```

```
(‘ P) ⌣? ∗ = yes (P⌣∗ P)

(‘ B) ⌣? (‘ B) = yes ⌣B

(‘ B) ⌣? (‘ (T₁ ⇒ T₂)) = no (λ ())

(‘ B) ⌣? (‘ (T₁ ⊗ T₂)) = no (λ ())

(‘ (T₁ ⇒ T₂)) ⌣? (‘ B) = no (λ ())

(‘ (T₁ ⇒ T₂)) ⌣? (‘ (T₃ ⇒ T₄)) = yes ⌣⇒

(‘ (T₁ ⇒ T₂)) ⌣? (‘ (T₃ ⊗ T₄)) = no (λ ())

(‘ (T₁ ⊗ T₂)) ⌣? (‘ B) = no (λ ())

(‘ (T₁ ⊗ T₂)) ⌣? (‘ (T₃ ⇒ T₄)) = no (λ ())

(‘ (T₁ ⊗ T₂)) ⌣? (‘ (T₃ ⊗ T₄)) = yes ⌣⊗


⌣trans : ∀ {P1 P2 P3} → (‘ P1) ⌣ (‘ P2) → (‘ P2) ⌣ (‘ P3) → (‘ P1) ⌣ (‘ P3)

⌣trans ⌣B ⌣B = ⌣B

⌣trans ⌣⇒ ⌣⇒ = ⌣⇒

⌣trans ⌣⊗ ⌣⊗ = ⌣⊗


⌣refl : ∀ T → T ⌣ T

⌣refl ∗ = ∗⌣∗

⌣refl (‘ B) = ⌣B

⌣refl (‘ (T₁ ⇒ T₂)) = ⌣⇒

⌣refl (‘ (T₁ ⊗ T₂)) = ⌣⊗


⌣sym : ∀ {S T} → S ⌣ T → T ⌣ S

⌣sym ∗⌣∗ = ∗⌣∗

⌣sym (∗⌣P P) = P⌣∗ P

⌣sym (P⌣∗ P) = ∗⌣P P

⌣sym ⌣B = ⌣B

⌣sym ⌣⇒ = ⌣⇒
```

```
⌣sym ⌣⊗ = ⌣⊗


⌣unique : ∀ {T1 T2}
  → (p1 p2 : T1 ⌣ T2)
  ---
  → p1 ≡ p2
⌣unique *⌣* *⌣* = refl
⌣unique (*⌣P P) (*⌣P .P) = refl
⌣unique (P⌣* P) (P⌣* .P) = refl
⌣unique ⌣B ⌣B = refl
⌣unique ⌣⇒ ⌣⇒ = refl
⌣unique ⌣⊗ ⌣⊗ = refl


-- shallow : ∀ {S T} → S ~ T → S ⌣ T
-- shallow *~* = *⌣*
-- shallow (*~P P) = *⌣P P
-- shallow (P~* P) = P⌣* P
-- shallow ~B = ⌣B
-- shallow (~⇒ p p₁) = ⌣⇒
-- shallow (~⊗ p p₁) = ⌣⊗


ground : PreType → PreType
ground B = B
ground (S ⇒ T) = * ⇒ *
ground (S ⊗ T) = * ⊗ *


ground-Ground : ∀ P → Ground (ground P)
ground-Ground B = `B
```

```
ground-Ground (S ⇒ T) = ‘⇒
ground-Ground (S ⊗ T) = ‘⊗

ground-⌣ : ∀ P → (‘ P) ⌣ (‘ (ground P))
ground-⌣ B = ⌣B
ground-⌣ (T₁ ⇒ T₂) = ⌣⇒
ground-⌣ (S ⊗ T) = ⌣⊗

_=?G_ : ∀ {P Q} → Ground P → Ground Q → Dec (P ≡ Q)
_=?G_ ‘B ‘B = yes refl
_=?G_ ‘B ‘⇒ = no (λ ())
_=?G_ ‘B ‘⊗ = no (λ ())
_=?G_ ‘⇒ ‘B = no (λ ())
_=?G_ ‘⇒ ‘⇒ = yes refl
_=?G_ ‘⇒ ‘⊗ = no (λ ())
_=?G_ ‘⊗ ‘B = no (λ ())
_=?G_ ‘⊗ ‘⇒ = no (λ ())
_=?G_ ‘⊗ ‘⊗ = yes refl

_=?GT_ : (G H : GroundType) → Dec (G ≡ H)
(P , gP) =?GT (Q , gQ) with gP =?G gQ
(P , gP) =?GT (Q , gQ) | yes refl rewrite ground-unique gP gQ = yes refl
(P , gP) =?GT (Q , gQ) | no ¬P≡Q  = no λ { refl → ¬P≡Q refl }

-- ground-≡ : ∀ {P Q} → Ground P → Ground Q → (‘ P) ⌣ (‘ Q) → P ≡ Q
-- ground-≡ ‘B ‘B P⌣Q = refl
-- ground-≡ ‘⇒ ‘⇒ P⌣Q = refl
-- ground-≡ ‘⊗ ‘⊗ P⌣Q = refl
```

```
-- ground-≢ : ∀ {P Q} → Ground P → Ground Q → ¬ (‘ P) ∪ (‘ Q) → ¬ (‘ P) ≡ (‘ Q)
-- ground-≢ ‘B ‘B ¬P∪Q = ⊥-elim (¬P∪Q ∪B)
-- ground-≢ ‘B ‘⇒ ¬P∪Q = λ ()
-- ground-≢ ‘B ‘⊗ ¬P∪Q = λ ()
-- ground-≢ ‘⇒ ‘B ¬P∪Q = λ ()
-- ground-≢ ‘⇒ ‘⇒ ¬P∪Q = λ _ → ¬P∪Q ∪⇒
-- ground-≢ ‘⇒ ‘⊗ ¬P∪Q = λ ()
-- ground-≢ ‘⊗ ‘B ¬P∪Q = λ ()
-- ground-≢ ‘⊗ ‘⇒ ¬P∪Q = λ ()
-- ground-≢ ‘⊗ ‘⊗ ¬P∪Q = λ _ → ¬P∪Q ∪⊗

¬∪-¬ground∪ : ∀ {P Q} → ¬ (‘ P) ∪ (‘ Q) → ¬ (‘ ground P) ∪ (‘ ground Q)
¬∪-¬ground∪ {B} {B} ¬P∪Q = λ _ → ¬P∪Q ∪B
¬∪-¬ground∪ {B} {S ⇒ T} ¬P∪Q = λ ()
¬∪-¬ground∪ {B} {S ⊗ T} ¬P∪Q = λ ()
¬∪-¬ground∪ {S ⇒ T} {B} ¬P∪Q = λ ()
¬∪-¬ground∪ {S ⇒ T} {S₁ ⇒ T₁} ¬P∪Q = λ _ → ¬P∪Q ∪⇒
¬∪-¬ground∪ {S ⇒ T} {S₁ ⊗ T₁} ¬P∪Q = λ ()
¬∪-¬ground∪ {S ⊗ T} {B} ¬P∪Q = λ ()
¬∪-¬ground∪ {S ⊗ T} {S₁ ⇒ T₁} ¬P∪Q = λ ()
¬∪-¬ground∪ {S ⊗ T} {S₁ ⊗ T₁} ¬P∪Q = λ _ → ¬P∪Q ∪⊗

-- matching
data _▷_ : Type → PreType → Set where
  same   : ∀ P → (‘ P) ▷ P
  coerce : ∀ P → Ground P → ∗ ▷ P
```

```
match-target : ∀ {T P} → T ▷ P → PreType
match-target {T} {P} m = P


match→consistency : ∀ {T P} → T ▷ P → T ~ (‘ P)
match→consistency (same P)      = ~refl (‘ P)
match→consistency (coerce P gP) = ∗~P P


meet : {S T : Type} → S ~ T → Type
meet ∗~∗ = ∗
meet (∗~P P) = (‘ P)
meet (P~∗ P) = (‘ P)
meet ~B = (‘ B)
meet (~⇒ S~T₁ S~T₂) = ‘ meet S~T₁ ⇒ meet S~T₂
meet (~⊗ S~T₁ S~T₂) = ‘ meet S~T₁ ⊗ meet S~T₂
```

A.2 GTLC

```
module GTLC
  (Label : Set)
  where
open import Types
open import Variables


infix  4 _ ⊢ _


data _ ⊢ _ : Context → Type → Set where


  var : ∀ {Γ T}
    → (x : Γ ∋ T)
```

```
  --------
  → Γ  ⊢  T


lam : ∀ {Γ S T}
  → (e : Γ , S  ⊢  T)
  -------------
  → Γ  ⊢  ‘ S ⇒ T


app : ∀ {Γ T1 S T T2}
  → (e1 : Γ  ⊢  T1)
  → (e2 : Γ  ⊢  T2)
  → (l  : Label)
  → (m  : T1 ▷ (S ⇒ T))
  → (T2~S : T2 ~ S)
  --------
  → Γ  ⊢  T


#t : ∀ {Γ}
  --------
  → Γ  ⊢  ‘ B


#f : ∀ {Γ}
  --------
  → Γ  ⊢  ‘ B


if : ∀ {Γ TB T1 T2}
  → Γ  ⊢  TB
  → Γ  ⊢  T1
```

```
    → Γ ⊢ T2
    → (l : Label)
    → (TB~B : TB ~ (‘ B))
    → (T1~T2 : T1 ~ T2)
    --------
    → Γ ⊢ (meet T1~T2)

  cons : ∀ {Γ T1 T2}
    → (e1 : Γ ⊢ T1)
    → (e2 : Γ ⊢ T2)
    -----
    → Γ ⊢ (‘ T1 ⊗ T2)

  car : ∀ {Γ S⊗T S T}
    → (e : Γ ⊢ S⊗T)
    → (l : Label)
    → (m : S⊗T ▷ (S ⊗ T))
    → Γ ⊢ S

  cdr : ∀ {Γ S⊗T S T}
    → (e : Γ ⊢ S⊗T)
    → (l : Label)
    → (m : S⊗T ▷ (S ⊗ T))
    → Γ ⊢ T

open import Terms Label
  renaming (_ ⊢ _ to _ ⊢ C_)
open import Cast Label
```

```
typeof : ∀ {Γ T} → Γ ⊢ T → Type
typeof {Γ} {T} e = T

-- the translation relation (in the form of function)

compile : ∀ {Γ T} → Γ ⊢ T → Γ ⊢ C T
compile (var x) = var x
compile (lam e) = lam (compile e)
compile (app e₁ e₂ l m T2~S)
  = app (compile e₁ ⟨ typeof e₁ ⟹[ l ] ‘ match-target m ⟩)
        (compile e₂ ⟨ typeof e₂ ⟹[ l ] _ ⟩)
compile #t = #t
compile #f = #t
compile (if e₁ e₂ e₃ l TB~B T1~T2)
  = if (compile e₁ ⟨ typeof e₁ ⟹[ l ] ‘ B ⟩)
       (compile e₂ ⟨ typeof e₂ ⟹[ l ] meet T1~T2 ⟩)
       (compile e₃ ⟨ typeof e₃ ⟹[ l ] meet T1~T2 ⟩)
compile (cons e₁ e₂) = cons (compile e₁) (compile e₂)
compile (car e l m)  = car (compile e ⟨ typeof e ⟹[ l ] (‘ match-target m) ⟩)
compile (cdr e l m)  = cdr (compile e ⟨ typeof e ⟹[ l ] (‘ match-target m) ⟩)
```

A.3 Casts in Source Language

```
module Cast
  (Label : Set)
  where

open import Types
```

```
data Cast : Type → Type → Set where
  _⟹[_]_ : (S : Type) → (l : Label) → (T : Type) → Cast S T
```

A.4 Variables

```
module Variables where


open import Types


data Context : Set where
  ∅   : Context
  _,_ : Context → Type → Context


infix  4 _∋_
infix 99 _,_


data _∋_ : Context → Type → Set where


  zero : ∀ {Γ A}
      ----------
    → Γ , A ∋ A


  suc  : ∀ {Γ A B}
    → Γ ∋ A
      ---------
    → Γ , B ∋ A


extL :  Type → Context → Context
```

```
extL t ∅ = ∅ , t

extL t (C , x) = (extL t C) , x
```

A.5 Expressions

```
module Terms

  (Label : Set)

  where


open import Types

open import Variables

open import Cast Label


infix  4 _ ⊢ _


data _ ⊢ _ : Context → Type → Set where


  var : ∀ {Γ T}

    → (x : Γ ∋ T)

    --------

    → Γ  ⊢  T


  lam : ∀ {Γ T1 T2}

    → (e : Γ , T1  ⊢  T2)

    -------------

    → Γ  ⊢  ‘ T1 ⇒ T2


  app : ∀ {Γ T1 T2}

    → (e1 : Γ  ⊢  ‘ T1 ⇒ T2)
```

```
  → (e2 : Γ ⊢ T1)
  --------
  → Γ ⊢ T2

#t : ∀ {Γ}
  --------
  → Γ ⊢ ‘ B

#f : ∀ {Γ}
  --------
  → Γ ⊢ ‘ B

if : ∀ {Γ T}
  → Γ ⊢ ‘ B
  → Γ ⊢ T
  → Γ ⊢ T
  --------
  → Γ ⊢ T

cons : ∀ {Γ T1 T2}
  → (e1 : Γ ⊢ T1)
  → (e2 : Γ ⊢ T2)
  -----
  → Γ ⊢ (‘ T1 ⊗ T2)

car : ∀ {Γ S T}
  → (e : Γ ⊢ (‘ S ⊗ T))
  → Γ ⊢ S
```

```
  cdr : ∀ {Γ S T}
    → (e : Γ ⊢ (‘ S ⊗ T))
    → Γ ⊢ T

  _⟨_⟩ : ∀ {Γ T S}
    → (e : Γ ⊢ S)
    → (c : Cast S T)
    --------
    → Γ ⊢ T

  -- blame : ∀ {Γ T}
  --   → (l : Label)
  --   ---
  --   → Γ ⊢ T
```

A.6 Observables

```
module Observables
  (Label : Set)
  where

open import Types

data ValueDisplay : Type → Set where
  dyn : ValueDisplay ∗

  #t : ValueDisplay (‘ B)
  #f : ValueDisplay (‘ B)
```

```
  lam : ∀ {T1 T2}
    ---
    → ValueDisplay (‘ T1 ⇒ T2)

  cons : ∀ {T1 T2}
    -- → ValueDisplay T1
    -- → ValueDisplay T2
      ---
    → ValueDisplay (‘ T1 ⊗ T2)

  -- inl : ∀ {T1 T2}
  --   ---
  --   → ValueDisplay (‘ T1 âŁ T2)

  -- inr : ∀ {T1 T2}
  --   ---
  --   → ValueDisplay (‘ T1 âŁ T2)

open import Error

Observable : Type → Set
Observable T = Error Label (ValueDisplay T)
```

A.7 The $\mathcal{R}(B)$ machine

```
open import R.BlameStrategies using (BlameStrategy)

module R.Machine
```

```
  (Label : Set)
  (BS : BlameStrategy Label)
  where
open BlameStrategy BS


open import Types
open import Variables using (∅)
open import Terms Label
open import Error
open import Observables Label
open import R.Values Label Injectable
open import Cast Label


open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong)


data Frame : Type → Type → Set where


  app₁ : ∀ {Γ S T}
    → (e2 : Γ ⊢ S)
    → (E : Env Γ)
    --------
    → Frame (‘ S ⇒ T) T


  app₂ : ∀ {S T}
    → (v1 : Value (‘ S ⇒ T))
    --------
    → Frame S T
  if₁ : ∀ {Γ T}
```

```
    → (e2 : Γ ⊢ T)
    → (e3 : Γ ⊢ T)
    → (E : Env Γ)
    ---------
    → Frame (‘ B) T

  cons₁ : ∀ {Γ T1 T2}
    → (e2 : Γ ⊢ T2)
    → (E : Env Γ)
    -----
    → Frame T1 (‘ T1 ⊗ T2)

  cons₂ : ∀ {T1 T2}
    → (v1 : Value T1)
    -----
    → Frame T2 (‘ T1 ⊗ T2)

  car₁ : ∀ {T1 T2} → Frame (‘ T1 ⊗ T2) T1
  cdr₁ : ∀ {T1 T2} → Frame (‘ T1 ⊗ T2) T2

  □⟨_⟩ : ∀ {S T}
    → (c : Cast S T)
    → Frame S T

data Cont : Type → Type → Set where

  [_]_ : ∀ {S T Z}
    → (F : Frame S T)
```

```
    → (k : Cont T Z)
    ---
    → Cont S Z

  □ : ∀ {Z}
    ----------
    → Cont Z Z

data OrdinaryState (T : Type) : Set where
  expr : ∀ {Γ S}
    → (e : Γ  ⊢  S)
    → (E : Env Γ)
    → (K : Cont S T)
    ------------
    → OrdinaryState T

  cont : ∀ {S}
    → (v : Value S)
    → (K : Cont S T)
    ------------
    → OrdinaryState T

  halt : (v : Value T) → OrdinaryState T

State : Type → Set
State T = Error Label (OrdinaryState T)

data Final {T : Type} : State T →  Set where
```

```
  halt : ∀ v
    → Final (return (halt v))


  error : ∀ l
    → Final (raise l)


data Progressing {T : Type} : State T →  Set where
  expr : ∀ {Γ S}
    → (e : Γ  ⊢  S)
    → (E : Env Γ)
    → (K : Cont S T)
    ------------
    → Progressing (return (expr e E K))


  cont : ∀ {S}
    → (v : Value S)
    → (K : Cont S T)
    ------------
    → Progressing (return (cont v K))


progressing-unique : ∀ {T} → {s : State T} → (sp1 sp2 : Progressing s) → sp1 ≡ sp2
progressing-unique (expr e E K) (expr .e .E .K) = refl
progressing-unique (cont v K) (cont .v .K) = refl


open import Data.Empty using (⊥; ⊥-elim)


final-progressing-absurd : ∀ {T} → {s : State T}
  → Final s
```

```
  → Progressing s
  → ⊥
final-progressing-absurd (halt v) ()
final-progressing-absurd (error l) ()


load : ∀ {T} → ∅ ⊢ T → State T
load e = return (expr e [] □)


do-app : ∀ {T1 T2 Z}
  → Value (‘ T1 ⇒ T2)
  → Value T1
  → Cont T2 Z
  → State Z
do-app (lam e E) u k
  = return (expr e (u :: E) k)
do-app (v f⟨(‘ T1 ⇒ T2) ⟹[ l ] (‘ T3 ⇒ T4)⟩) u k
  = return (cont u ([ □⟨ T3 ⟹[ l ] T1 ⟩ ]
                    [ app₂ v ]
                    [ □⟨ T2 ⟹[ l ] T4 ⟩ ]
                    k))


do-car : ∀ {T1 T2 Z}
  → Value (‘ T1 ⊗ T2)
  → Cont T1 Z
  → State Z
do-car (cons v1 v2) k = return (cont v1 k)
do-car (v p⟨ (‘ T1 ⊗ T2) ⟹[ l ] (‘ T3 ⊗ T4) ⟩) k
  = return (cont v ([ car₁ ] [ □⟨ T1 ⟹[ l ] T3 ⟩ ] k))
```

```
do-cdr : ∀ {T1 T2 Z}
  → Value (‘ T1 ⊗ T2)
  → Cont T2 Z
  → State Z
do-cdr (cons v1 v2) k = return (cont v2 k)
do-cdr (v p⟨ (‘ T1 ⊗ T2) ⟹[ l ] (‘ T3 ⊗ T4) ⟩) k
  = return (cont v ([ cdr₁ ] [ □⟨ T2 ⟹[ l ] T4 ⟩ ] k))


do-cast : ∀ {T1 T2 Z}
  → Cast T1 T2
  → Value T1
  → Cont T2 Z
  → State Z
do-cast c v k = ⟦ c ⟧ v >>= λ u → return (cont u k)


cnd : {A : Set} → Value (‘ B) → (x y : A) → A
cnd #t x y = x
cnd #f x y = y


apply-cont : ∀ {T1 T2}
  → Value T1
  → Cont T1 T2
  ---
  → State T2
apply-cont v ([ app₁ e2 E ] k) = return (expr e2 E ([ app₂ v ] k))
apply-cont v ([ app₂ v1 ] k) = do-app v1 v k
apply-cont v ([ if₁ e2 e3 E ] k) = return (expr (cnd v e2 e3) E k)
```

```
apply-cont v ([ cons₁ e2 E ] k) = return (expr e2 E ([ cons₂ v ] k))
apply-cont v ([ cons₂ v1 ] k) = return (cont (cons v1 v) k)
apply-cont v ([ car₁ ] k) = do-car v k
apply-cont v ([ cdr₁ ] k) = do-cdr v k
apply-cont v ([ □⟨ c ⟩ ] k) = do-cast c v k
apply-cont v □ = return (halt v)

-- reduction
progress : ∀ {Z} → {s : State Z} → Progressing s → State Z
progress (expr (var x) E K)       = return (cont (lookup E x) K)
progress (expr (lam e) E K)       = return (cont (lam e E) K)
progress (expr (app e1 e2) E K)   = return (expr e1 E ([ app₁ e2 E ] K))
progress (expr #t E K)            = return (cont #t K)
progress (expr #f E K)            = return (cont #f K)
progress (expr (if e1 e2 e3) E K) = return (expr e1 E ([ if₁ e2 e3 E ] K))
progress (expr (cons e1 e2) E K)  = return (expr e1 E (([ cons₁ e2 E ] K)))
progress (expr (car e) E K)       = return (expr e E ([ car₁ ] K))
progress (expr (cdr e) E K)       = return (expr e E ([ cdr₁ ] K))
progress (expr (e ⟨ c ⟩) E K)     = return (expr e E ([ □⟨ c ⟩ ] K))
progress (cont v k)               = apply-cont v k

data _⟶_ {T : Type} : State T → State T → Set where
  it : ∀ {s}
    → (sp : Progressing s)
    → s ⟶ progress sp

open import Bisimulation.Bisimulation using (System)
```

```
deterministic : ∀ {T}
  → {s t1 t2 : State T}
  → s ⟶ t1
  → s ⟶ t2
  → t1 ≡ t2
deterministic (it sp1) (it sp2) = cong progress (progressing-unique sp1 sp2)


system : ∀ {T} → System (State T)
system = record
           { _⟶_ = _⟶_
           ; Final = Final
           ; final-progressing-absurd = λ { sf (it sp) → final-progressing-absurd sf sp }
           ; deterministic = deterministic
           }


module Eval {T : Type} where
  open import Data.Product using (∃-syntax)
  open System (system {T}) using (_⟶*_; []; _∷_; _⟶+_; _++_) public


  observe : Value T → ValueDisplay T
  observe (dyn I v) = dyn
  observe #t = #t
  observe #f = #f
  observe (lam e E) = lam
  observe (u f⟨ c ⟩) = lam
  observe (cons v₁ v₂) = cons
  observe (u p⟨ c ⟩) = cons
```

```
  data Evalo (e : ∅ ⊢ T) : Observable T → Set where
    val : ∀ {v}
      → ((load e) ⟶* return (halt v))
      → Evalo e (return (observe v))
    err : ∀ {l}
      → ((load e) ⟶* raise l)
      → Evalo e (raise l)

open Eval public
```

A.8 Values of $\mathcal{R}(B)$

```
open import Types

module R.Values
  (Label : Set)
  (Injectable : PreType → Set)
  where

open import Terms Label
open import Variables
open import Cast Label using (Cast; _⟹[_]_)
open import Error

open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Relation.Nullary using (¬_)

mutual
```

```
data Value : Type → Set where
  dyn : ∀ {P}
    → (I : Injectable P)
    → (v : Value (‘ P))
    ---
    → Value ∗

  #t : Value (‘ B)
  #f : Value (‘ B)

  -- unit :
  --   --------
  --     Value (‘ U)

  lam : ∀ {Γ T1 T2}
    → (e : Γ , T1  ⊢  T2)
    → (E : Env Γ)
    -------------
    → Value (‘ T1 ⇒ T2)

  _f⟨_⟩ : ∀ {T1 T2 T3 T4}
    → (v : Value (‘ T1 ⇒ T2))
    → (c : Cast (‘ T1 ⇒ T2) (‘ T3 ⇒ T4))
    -----
    → Value (‘ T3 ⇒ T4)

  cons : ∀ {T1 T2}
    → (v1 : Value T1)
```

```
      → (v2 : Value T2)
      ------
      → Value (‘ T1 ⊗ T2)

    _p⟨_⟩ : ∀ {T1 T2 T3 T4}
      → (v : Value (‘ T1 ⊗ T2))
      → (c : Cast (‘ T1 ⊗ T2) (‘ T3 ⊗ T4))
      -----
      → Value (‘ T3 ⊗ T4)



  data Env : Context → Set where
    []  : Env ∅
    _::_ : ∀ {Γ T}
      → (v : Value T)
      → Env Γ
      → Env (Γ , T)

lookup : ∀ {Γ T} → Env Γ → Γ ∋ T → Value T
lookup (c :: E) zero = c
lookup (c :: E) (suc n) = lookup E n

open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Relation.Nullary using (yes; no)

add-proxy : ∀ {P Q} → Value (‘ P) → Cast (‘ P) (‘ Q) → (‘ P) ⌣ (‘ Q) → Value (‘ Q)
add-proxy v ((‘ B) ⟹[ l ] (‘ B)) ⌣B = v
```

```
add-proxy v ((‘ (T1 ⇒ T2)) ⟹[ l ] (‘ (T3 ⇒ T4))) ⌣⇒
  = v f⟨(‘ T1 ⇒ T2) ⟹[ l ] (‘ T3 ⇒ T4)⟩
add-proxy v ((‘ (T1 ⊗ T2)) ⟹[ l ] (‘ (T3 ⊗ T4))) ⌣⊗
  = v p⟨(‘ T1 ⊗ T2) ⟹[ l ] (‘ T3 ⊗ T4)⟩
```

A.9 Blame Strategies

```
module R.BlameStrategies
  (Label : Set)
  where

open import Types
open import R.Values Label
open import Cast Label
open import Error using (Error; return; raise; _>>=_)

open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Data.Empty using (⊥-elim)

record BlameStrategy : Set₁ where
  field
    Injectable : PreType → Set
    ⟦_⟧ : ∀ {S T}
      → Cast S T
      -----
      → Value Injectable S
      → Error Label (Value Injectable T)

module LazyD where
```

```
  open import Types
  open import Relation.Nullary using (yes; no)

  I? : PreType → Set
  I? = Same

  apply-cast' : ∀ {P Q} → Value I? (‘ P) → Cast (‘ P) (‘ Q)
    → Error Label (Value I? (‘ Q))
  apply-cast' v ((‘ P) ⟹[ l ] (‘ Q)) with (‘ P) ⌣? (‘ Q)
  ... | yes P⌣Q = return (add-proxy I? v ((‘ P) ⟹[ l ] (‘ Q)) P⌣Q)
  ... | no ¬P⌣Q = raise l

  ⟦_⟧ : ∀ {S T} → Cast S T → Value I? S → Error Label (Value I? T)
  ⟦ ∗ ⟹[ l ] ∗ ⟧ v = return v
  ⟦ ∗ ⟹[ l ] (‘ Q) ⟧ (dyn (same P) v) = apply-cast' v ((‘ P) ⟹[ l ] (‘ Q))
  ⟦ (‘ P) ⟹[ l ] ∗ ⟧ v = return (dyn (same P) v)
  ⟦ (‘ P) ⟹[ l ] (‘ Q) ⟧ v = apply-cast' v ((‘ P) ⟹[ l ] (‘ Q))

LazyDBS : BlameStrategy
LazyDBS = record { Injectable = LazyD.I? ; ⟦_⟧ = LazyD.⟦_⟧ }

module LazyUD where
  open import Types

  I? : PreType → Set
  I? = Ground

  open import Relation.Nullary using (yes; no)
```

```
  open import Relation.Binary.PropositionalEquality using (refl)


  project : Value I? ⋆ → Label → {Q : PreType} → Ground Q
    → Error Label (Value I? (‘ Q))
  project (dyn gP v) l gQ with (‘ unground gP) =? (‘ unground gQ)
  ... | yes refl = return v
  ... | no ¬p = raise l


  ⟦_⟧ : ∀ {S T} → Cast S T → Value I? S → Error Label (Value I? T)
  ⟦ ⋆   ⟹[ l ] ⋆ ⟧ v = return v
  ⟦ ‘ P ⟹[ l ] ⋆ ⟧ v with ground? P
  ... | yes Pg = return (dyn Pg v)
  ... | no ¬Pg = return (dyn (ground-Ground P)
                             (add-proxy I? v ((‘ P) ⟹[ l ] (‘ ground P))
                                             (ground-⌣ P)))
  ⟦ ‘ P ⟹[ l ] ‘ Q ⟧ v with (‘ P) ⌣? (‘ Q)
  ... | yes p = return (add-proxy I? v ((‘ P) ⟹[ l ] (‘ Q)) p)
  ... | no ¬p = raise l
  ⟦ ⋆ ⟹[ l ] ‘ Q ⟧ v with ground? Q
  ⟦ ⋆ ⟹[ l ] ‘ Q ⟧ v | yes Qg = project v l Qg
  ⟦ ⋆ ⟹[ l ] ‘ Q ⟧ v | no ¬Qg
    = project v l (ground-Ground Q) >>= λ u →
      return (add-proxy I? u (‘ ground Q ⟹[ l ] ‘ Q) (⌣sym (ground-⌣ Q)))


LazyUDBS : BlameStrategy
LazyUDBS = record { Injectable = LazyUD.I? ; ⟦_⟧ = LazyUD.⟦_⟧ }
```

Appendix B

Cast ADT And $\mathcal{S}(C)$

B.1 Cast ADT

```
open import Types


module S.CastADT
  (Label : Set)
  (Injectable : PreType → Set)
  where


open import Error using (_>=>_; Error; return)
open import Cast Label renaming (Cast to SrcCast)
import S.Values using (Value)


open import Relation.Binary.PropositionalEquality using (_≡_)


record CastADT : Set₁ where
  open S.Values Label Injectable
  field
    Cast : Type → Type → Set


    id : (T : Type) → Cast T T
    ⌈_⌉ : ∀ {S T} → SrcCast S T → Cast S T
    _⨟_ : ∀ {T1 T2 T3}
      → Cast T1 T2
      → Cast T2 T3
```

```
      -----
      → Cast T1 T3
    ⟦_⟧ : ∀ {T1 T2}
      → Cast T1 T2
      -----
      → Value Cast T1
      → Error Label (Value Cast T2)
    lem-id : ∀ T v
      → ⟦ id T ⟧ v ≡ return v
    lem-seq : ∀ {T1 T2 T3}
      → (c1 : Cast T1 T2)
      → (c2 : Cast T2 T3)
      → ∀ v
      --------------------
      → ⟦ c1 ⨟ c2 ⟧ v ≡ (⟦ c1 ⟧ >=> ⟦ c2 ⟧) v
```

B.2 Lazy D Cast ADT

```
module S.LazyDCastADT
  (Label : Set)
  where


open import Types


open import Variables
open import Terms Label using (_ ⊢ _)
open import Error
open import Cast Label using (_⟹[_]_)
```

```
open import R.BlameStrategies Label using (BlameStrategy; LazyDBS)
open BlameStrategy LazyDBS using (Injectable)

open import S.CastADT Label Injectable
import S.Values using (Env; Value; dyn; #t; #f; lam⟨_⇒_⟩; cons⟨_⊗_⟩)

open import Relation.Binary.PropositionalEquality using (_≡_)
open import Relation.Nullary using (¬_; yes; no)

record LazyD (ADT : CastADT) : Set where
  open CastADT ADT
  open S.Values Label Injectable Cast
  field
    eq-¬⌣ : ∀ {T1 T2}
      → (l : Label)
      → ¬ (T1 ⌣ T2)
      → (v : Value T1)
      ---
      → ⟦ ⌈ T1 ⟹[ l ] T2 ⌉ ⟧ v
          ≡
        raise l

    eq-** : ∀ l
      → (v : Value ∗)
      → ⟦ ⌈ ∗ ⟹[ l ] ∗ ⌉ ⟧ v
          ≡
        return v
```

```
eq-P∗ : ∀ {P}
  → (l : Label)
  → (v : Value (‘ P))
  → ⟦ ⌈ (‘ P) ⟹[ l ] ∗ ⌉ ⟧ v
      ≡
    return (dyn (same P) v)

eq-∗P : ∀ Q P l v
  → ⟦ ⌈ ∗ ⟹[ l ] (‘ Q) ⌉ ⟧ (dyn (same P) v)
      ≡
    ⟦ ⌈ (‘ P) ⟹[ l ] (‘ Q) ⌉ ⟧ v

eq-B : ∀ l v
  → ⟦ ⌈ (‘ B) ⟹[ l ] (‘ B) ⌉ ⟧ v
      ≡
    return v

eq-⇒ : ∀ T21 T22 T11 T12
  → ∀ {S T}
  → (l : Label)
  → {Γ : Context}
  → (c₁ : Cast T11 S)
  → (c₂ : Cast T T12)
  → (e : (Γ , S) ⊢ T)
  → (E : Env Γ)
  → ⟦ ⌈ (‘ T11 ⇒ T12) ⟹[ l ] (‘ T21 ⇒ T22) ⌉ ⟧ (lam⟨ c₁ ⇒ c₂ ⟩ e E)
      ≡
    return (lam⟨ ⌈ T21 ⟹[ l ] T11 ⌉ ⨟ c₁ ⇒ c₂ ⨟ ⌈ T12 ⟹[ l ] T22 ⌉ ⟩ e E)
```

```
  eq-⊗ : ∀ T21 T22 T11 T12
    → ∀ {S T}
    → (l : Label)
    → (c₁ : Cast S T11)
    → (c₂ : Cast T T12)
    → (v1 : Value S)
    → (v2 : Value T)
    → ⟦ ⌈ (‘ T11 ⊗ T12) ⟹[ l ] (‘ T21 ⊗ T22) ⌉ ⟧ (cons⟨ c₁ ⊗ c₂ ⟩ v1 v2)
        ≡
      return (cons⟨ c₁ ⨟ ⌈ T11 ⟹[ l ] T21 ⌉ ⊗ c₂ ⨟ ⌈ T12 ⟹[ l ] T22 ⌉ ⟩ v1 v2)
```

B.3 Lazy UD Cast ADT

```
module S.LazyUDCastADT
  (Label : Set)
  where

open import Types

open import Variables
open import Terms Label using (_ ⊢ _)
open import Error
open import Cast Label using (_⟹[_]_)

open import R.BlameStrategies Label using (BlameStrategy; LazyUDBS)
open BlameStrategy LazyUDBS using (Injectable)

open import S.CastADT Label Injectable
```

```
import S.Values using (Env; Value; dyn; #t; #f; lam⟨_⇒_⟩; cons⟨_⊗_⟩)


open import Relation.Binary.PropositionalEquality using (_≡_)
open import Relation.Nullary using (¬_; yes; no)


record LazyUD (ADT : CastADT) : Set where
  open CastADT ADT
  open S.Values Label Injectable Cast
  field
    eq-¬⌣ : ∀ {T1 T2}
      → (v : Value T1)
      → (l : Label)
      → ¬ (T1 ⌣ T2)
      ---
      → ⟦ ⌈ T1 ⟹[ l ] T2 ⌉ ⟧ v
          ≡
        raise l


    eq-** : ∀ l
      → (v : Value ∗)
      → ⟦ ⌈ ∗ ⟹[ l ] ∗ ⌉ ⟧ v
          ≡
        return v


    eq-P∗ : ∀ {P}
      → (v : Value (‘ P))
      → (l : Label)
      → ¬ Ground P
```

```
  → ⟦ ⌈ (‘ P) ⟹[ l ] ∗ ⌉ ⟧ v
      ≡
    ⟦ ⌈ (‘ P) ⟹[ l ] (‘ ground P) ⌉ ⟧ v >>= ⟦ ⌈ (‘ ground P) ⟹[ l ] ∗ ⌉ ⟧


eq-I∗ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → ⟦ ⌈ ‘ P ⟹[ l ] ∗ ⌉ ⟧ v
      ≡
    return (dyn gP v)


eq-∗P : ∀ {P}
  → (v : Value ∗)
  → (l : Label)
  → ¬ Ground P
  → ⟦ ⌈ ∗ ⟹[ l ] (‘ P) ⌉ ⟧ v
      ≡
    ⟦ ⌈ ∗ ⟹[ l ] (‘ ground P) ⌉ ⟧ v >>= ⟦ ⌈ (‘ ground P) ⟹[ l ] (‘ P) ⌉ ⟧


eq-∗I-succ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → ⟦ ⌈ ∗ ⟹[ l ] (‘ P) ⌉ ⟧ (dyn gP v)
      ≡
    return v
```

```
eq-∗I-fail : ∀ {P Q}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → (gQ : Ground Q)
  → ¬ (‘ P) ≡ (‘ Q)
  → ⟦ ⌈ ∗ ⟹[ l ] (‘ Q) ⌉ ⟧ (dyn gP v)
      ≡
    raise l


eq-B : ∀ l b
  → ⟦ ⌈ (‘ B) ⟹[ l ] (‘ B) ⌉ ⟧ b
      ≡
    return b


eq-⇒ : ∀ T21 T22 T11 T12
  → ∀ {S T}
  → (l : Label)
  → {Γ : Context}
  → (c₁ : Cast T11 S)
  → (c₂ : Cast T T12)
  → (e : (Γ , S) ⊢ T)
  → (E : Env Γ)
  → ⟦ ⌈ (‘ T11 ⇒ T12) ⟹[ l ] (‘ T21 ⇒ T22) ⌉ ⟧ (lam⟨ c₁ ⇒ c₂ ⟩ e E)
      ≡
    return (lam⟨ ⌈ T21 ⟹[ l ] T11 ⌉ ⨟ c₁ ⇒ c₂ ⨟ ⌈ T12 ⟹[ l ] T22 ⌉ ⟩ e E)


eq-⊗ : ∀ T21 T22 T11 T12
```

```
      → ∀ {S T}
      → (l : Label)
      → (c₁ : Cast S T11)
      → (c₂ : Cast T T12)
      → (v1 : Value S)
      → (v2 : Value T)
      → ⟦ ⌈ (‘ T11 ⊗ T12) ⟹[ l ] (‘ T21 ⊗ T22) ⌉ ⟧ (cons⟨ c₁ ⊗ c₂ ⟩ v1 v2)
          ≡
        return (cons⟨ c₁ ⨟ ⌈ T11 ⟹[ l ] T21 ⌉ ⊗ c₂ ⨟ ⌈ T12 ⟹[ l ] T22 ⌉ ⟩ v1 v2)
```

B.4 The $\mathcal{S}(C)$ Machine

```
open import Types
open import S.CastADT

module S.Machine
  (Label : Set)
  (Injectable : PreType → Set)
  (cast-adt : CastADT Label Injectable)
  where

open CastADT cast-adt using (Cast; id; ⌈_⌉; _⨟_; ⟦_⟧)

open import Error

open import Variables using (∅)
open import Cast Label using (_⟹[_]_)
open import Terms Label
open import Observables Label using (Observable; ValueDisplay; dyn; #t; #f; lam; cons)
```

```
open import S.Values Label Injectable Cast


open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong)


data Frame : Type → Type → Set where


  app₁ : ∀ {Γ S T}
    → (e2 : Γ ⊢ S)
    → (E : Env Γ)
    --------
    → Frame (‘ S ⇒ T) T


  app₂ : ∀ {S T}
    → (v1 : Value (‘ S ⇒ T))
    --------
    → Frame S T


  if₁ : ∀ {Γ T}
    → (e2 : Γ ⊢ T)
    → (e3 : Γ ⊢ T)
    → (E : Env Γ)
    ---------
    → Frame (‘ B) T


  cons₁ : ∀ {Γ T1 T2}
    → (e2 : Γ ⊢ T2)
    → (E : Env Γ)
    -----
```

```
    → Frame T1 (‘ T1 ⊗ T2)


  cons₂ : ∀ {T1 T2}
    → (v1 : Value T1)
    -----
    → Frame T2 (‘ T1 ⊗ T2)


  car₁ : ∀ {T1 T2} → Frame (‘ T1 ⊗ T2) T1
  cdr₁ : ∀ {T1 T2} → Frame (‘ T1 ⊗ T2) T2


mutual
  data PreCont : Type → Type → Set where


    [_]_ : ∀ {R S T}
      → (F : Frame R S)
      → (k : Cont S T)
      ---
      → PreCont R T


    □ : ∀ {Z}
      ----------
      → PreCont Z Z


  data Cont (T1 T2 : Type) : Set where
    [□⟨_⟩]_ : ∀ {T}
      → (c : Cast T1 T)
      → (k : PreCont T T2)
      → Cont T1 T2
```

```
  -- record Cont (T1 T2 : Type) : Set where
  --   inductive
  --   constructor [□⟨_⟩]_
  --   field
  --     {T} : Type
  --     c : Cast T1 T
  --     k : PreCont T T2


ext-cont : ∀ {T1 T2 T3} → Cast T1 T2 → Cont T2 T3 → Cont T1 T3
-- ext-cont c (‘ k)     = [□⟨ c ⟩] k
ext-cont c ([□⟨ d ⟩] k) = [□⟨ c ⨟ d ⟩] k


data OrdinaryState (Z : Type) : Set where
  expr : ∀ {Γ T}
    → (e : Γ ⊢ T)
    → (E : Env Γ)
    → (K : Cont T Z)
    ------------
    → OrdinaryState Z


  cont : ∀ {T}
    → (v : Value T)
    → (K : Cont T Z)
    ------------
    → OrdinaryState Z


  halt : (v : Value Z) → OrdinaryState Z
```

```
State : Type → Set
State T = Error Label (OrdinaryState T)

data Final {Z : Type} : State Z →  Set where
  halt : ∀ v
    → Final (return (halt v))

  error : ∀ l
    → Final (raise l)

data Progressing {Z : Type} : State Z →  Set where
  expr : ∀ {Γ T}
    → (e : Γ  ⊢  T)
    → (E : Env Γ)
    → (K : Cont T Z)
    ------------
    → Progressing (return (expr e E K))

  cont : ∀ {T}
    → (v : Value T)
    → (k : Cont T Z)
    ------------
    → Progressing (return (cont v k))

progressing-unique : ∀ {T} → {s : State T} → (sp1 sp2 : Progressing s) → sp1 ≡ sp2
progressing-unique (expr e E K) (expr .e .E .K) = refl
progressing-unique (cont v k) (cont .v .k) = refl
```

```
open import Data.Empty using (⊥; ⊥-elim)


final-progressing-absurd : ∀ {T} → {s : State T}
  → Final s
  → Progressing s
  → ⊥
final-progressing-absurd (halt v) ()
final-progressing-absurd (error l) ()


load : ∀ {T} → ∅ ⊢ T → State T
load e = return (expr e [] ([□⟨ id _ ⟩] □))


do-app : ∀ {T1 T2 Z}
  → Value (‘ T1 ⇒ T2)
  → Value T1
  → Cont T2 Z
  → State Z
-- do-app (lam e E) v K
--   = return (expr e (v :: E) K)
do-app (lam⟨ c ⇒ d ⟩ e E) v K
  = ⟦ c ⟧ v >>= λ u →
    return (expr e (u :: E) (ext-cont d K))


do-car : ∀ {T1 T2 Z}
  → Value (‘ T1 ⊗ T2)
  → Cont T1 Z
  → State Z
```

```
do-car (cons⟨ c1 ⊗ c2 ⟩ v1 v2) k = ⟦ c1 ⟧ v1 >>= λ v' → return (cont v' k)


do-cdr : ∀ {T1 T2 Z}
  → Value (‘ T1 ⊗ T2)
  → Cont T2 Z
  → State Z
do-cdr (cons⟨ c1 ⊗ c2 ⟩ v1 v2) k = ⟦ c2 ⟧ v2 >>= λ v' → return (cont v' k)


cnd : {A : Set} → Value (‘ B) → (x y : A) → A
cnd #t x y = x
cnd #f x y = y


mk-cont : ∀ {S T} → PreCont S T → Cont S T
mk-cont k = [□⟨ id _ ⟩] k


apply-cont : ∀ {T1 T2}
  → Value T1
  → PreCont T1 T2
  → State T2
apply-cont v ([ app₁ e2 E ] k) = return (expr e2 E (mk-cont ([ app₂ v ] k)))
apply-cont v ([ app₂ v1 ] k) = do-app v1 v k
apply-cont v ([ if₁ e2 e3 E ] k) = return (expr (cnd v e2 e3) E k)
apply-cont v ([ cons₁ e2 E ] k) = return (expr e2 E (mk-cont ([ cons₂ v ] k)))
apply-cont v ([ cons₂ v1 ] k) = return (cont (cons⟨ id _ ⊗ id _ ⟩ v1 v) k)
apply-cont v ([ car₁ ] k) = do-car v k
apply-cont v ([ cdr₁ ] k) = do-cdr v k
apply-cont v □ = return (halt v)
```

```
progress : ∀ {Z} → {s : State Z} → Progressing s → State Z
progress (expr (var x) E K)       = return (cont (lookup E x) K)
progress (expr #t E K)            = return (cont #t K)
progress (expr #f E K)            = return (cont #f K)
progress (expr (if e1 e2 e3) E K) = return (expr e1 E (mk-cont ([ if₁ e2 e3 E ] K)))
progress (expr (lam e) E K)       = return (cont (lam⟨ id _ ⇒ id _ ⟩ e E)  K)
progress (expr (app e1 e2) E K)   = return (expr e1 E (mk-cont ([ app₁ e2 E ] K)))
progress (expr (cons e1 e2) E K)  = return (expr e1 E (mk-cont ([ cons₁ e2 E ] K)))
progress (expr (car e) E K)       = return (expr e E (mk-cont ([ car₁ ] K)))
progress (expr (cdr e) E K)       = return (expr e E (mk-cont ([ cdr₁ ] K)))
progress (expr (e ⟨ c ⟩) E K)     = return (expr e E (ext-cont ⌈ c ⌉ K))
-- progress (expr (blame l) E K)     = raise l
progress (cont v ([□⟨ c ⟩] k)) = ⟦ c ⟧ v >>= λ v' → apply-cont v' k


data _⟶_ {T : Type} : State T → State T → Set where
  it : ∀ {s}
    → (sp : Progressing s)
    → s ⟶ progress sp


open import Bisimulation.Bisimulation using (System)


deterministic : ∀ {T}
  → {s t1 t2 : State T}
  → s ⟶ t1
  → s ⟶ t2
  → t1 ≡ t2
deterministic (it sp1) (it sp2) = cong progress (progressing-unique sp1 sp2)
```

```
system : ∀ {T} → System (State T)
system = record
           { _⟶_ = _⟶_
           ; Final = Final
           ; final-progressing-absurd =
             λ { sf (it sp) → final-progressing-absurd sf sp }
           ; deterministic = deterministic
           }


module Eval {T : Type} where
  open import Data.Product using (∃-syntax)
  open System (system {T}) using (_⟶*_; []; _∷_; _⟶+_; _++_) public


  observe : Value T → ValueDisplay T
  observe (dyn Pi v) = dyn
  observe #t = #t
  observe #f = #f
  observe (lam⟨ c1 ⇒ c2 ⟩ e E) = lam
  observe (cons⟨ c1 ⊗ c2 ⟩ v1 v2) = cons


  data Evalo (e : ∅ ⊢ T) : Observable T → Set where
    val : ∀ {v} → (load e) ⟶* return (halt v) → Evalo e (return (observe v))
    err : ∀ {l} → (load e) ⟶* raise l → Evalo e (raise l)


open Eval public
```

B.5 Values in $\mathcal{S}(C)$

```
open import Types
```

```
module S.Values
  (Label : Set)
  (Injectable : PreType → Set)
  (Cast : Type → Type → Set)
  where

open import Relation.Binary.PropositionalEquality using (_≡_; refl)

open import Terms Label
open import Variables

mutual

  data Value : Type → Set where
    dyn : ∀ {P}
      → (Pi : Injectable P)
      → (v  : Value (‘ P))
      ---
      → Value ⋆

    #t :
      --------
        Value (‘ B)

    #f :
      --------
        Value (‘ B)
```

```
-- lam : ∀ {Γ T1 T2}
--   → (e : (Γ , T1)  ⊢  T2)
--   → (E : Env Γ)
--   -------------
--   → Value (‘ T1 ⇒ T2)

lam⟨_⇒_⟩ : ∀ {Γ T1 T2 T3 T4}
  → (c1 : Cast T3 T1)
  → (c2 : Cast T2 T4)
  → (e : (Γ , T1)  ⊢  T2)
  → (E : Env Γ)
  -------------
  → Value (‘ T3 ⇒ T4)

-- cons : ∀ {T1 T2}
--   → x(v1 : Value T1)
--   → (v2 : Value T2)
--   ------
--   → Value (‘ T1 ⊗ T2)

cons⟨_⊗_⟩ : ∀ {T1 T2 T3 T4}
  → (c1 : Cast T1 T3)
  → (c2 : Cast T2 T4)
  → (v1 : Value T1)
  → (v2 : Value T2)
  -------------
  → Value (‘ T3 ⊗ T4)
```

```
  data Env : Context → Set where
    []  : Env ∅
    _::_ : ∀ {Γ T}
      → (v : Value T)
      → Env Γ
      → Env (Γ , T)


lookup : ∀ {Γ T} → Env Γ → Γ ∋ T → Value T
lookup (c :: E) zero = c
lookup (c :: E) (suc x) = lookup E x
```

Appendix C

Bisimulate $\mathcal{R}(B)$ and $\mathcal{S}(C)$

C.1 Bisimulation Relation

```
open import Types
open import R.BlameStrategies
open import S.CastADT


module Bisimulation.BisimulationRelation
  (Label : Set)
  (BS : BlameStrategy Label)
  (CADT : CastADT Label (BlameStrategy.Injectable BS))
  where


open BlameStrategy BS using (Injectable) public


open import Variables
open import Types
open import Terms Label
open import Error
open import Cast Label using () renaming (Cast to SCast)
open import Observables Label


module L where
  open BlameStrategy BS using (⟦_⟧) public
  open import R.Values Label Injectable public
  open import Cast Label public
```

```
  open import R.Machine Label BS public


open L hiding (_++_; lookup; _⟶_; _⟶*_; _⟶+_)


module R where
  open import S.Values Label Injectable (CastADT.Cast CADT) public
  open CastADT CADT public
  open import S.Machine Label Injectable CADT public


open R hiding (_++_; lookup; id; _⨟_; _⟶_; _⟶*_; _⟶+_; [□⟨_⟩]_)
  renaming (Cast to DCast)


data CastList : Type → Type → Set where
  [] : ∀ {T}
    → CastList T T


  _::_ : ∀ {T1 T2 T3}
    → L.Cast T1 T2
    → CastList T2 T3
    → CastList T1 T3


_++_ : ∀ {T1 T2 T3} → CastList T1 T2 → CastList T2 T3 → CastList T1 T3
[] ++ ds    = ds
(c :: cs) ++ ds = c :: (cs ++ ds)


data CastListRelate : {S T : Type} → CastList S T → R.Cast S T → Set where
  id : ∀ {T}
    → CastListRelate {T} {T} [] (R.id T)
```

```
  just : ∀ {T1 T2}
    → (c : L.Cast T1 T2)
    → CastListRelate (c ∷ []) R.⌈ c ⌉

  _⨟_ : ∀ {T1 T2 T3}
    → {lcs : CastList  T1 T2}
    → {rcs : R.Cast T1 T2}
    → (cs : CastListRelate lcs rcs)
    → {lds : CastList  T2 T3}
    → {rds : R.Cast T2 T3}
    → (ds : CastListRelate lds rds)
    -----
    → CastListRelate (lcs ++ lds) (rcs R.⨟ rds)

lcast : ∀ {T1 T2 lc rc} → CastListRelate {T1} {T2} lc rc → CastList T1 T2
lcast {lc = lc} c = lc

rcast : ∀ {T1 T2 lc rc} → CastListRelate {T1} {T2} lc rc → R.Cast T1 T2
rcast {rc = rc} c = rc

data FCastList : Type → Type → Type → Type → Set where
  [] : ∀ {T11 T12}
    → FCastList T11 T12 T11 T12
  _≪_ : ∀ {T11 T12 T21 T22 T31 T32}
    → FCastList T11 T12 T21 T22
    → L.Cast (‘ T21 ⇒ T22) (‘ T31 ⇒ T32)
    → FCastList T11 T12 T31 T32
```

```
view-lambda : ∀ {T11 T12 T21 T22}
  → L.Value (‘ T11 ⇒ T12)
  → FCastList T11 T12 T21 T22
  → L.Value (‘ T21 ⇒ T22)
view-lambda v [] = v
view-lambda v (cs ≪ c) = view-lambda v cs f⟨ c ⟩

dom : ∀ {T11 T12 T21 T22} → FCastList T11 T12 T21 T22 → CastList T21 T11
dom [] = []
dom (cs ≪ ((‘ T21 ⇒ T22) ⟹[ l ] (‘ T31 ⇒ T32)))
  = ( T31 ⟹[ l ] T21 ) ∷ dom cs

cod : ∀ {T11 T12 T21 T22} → FCastList T11 T12 T21 T22 → CastList T12 T22
cod [] = []
cod (cs ≪ ((‘ T21 ⇒ T22) ⟹[ l ] (‘ T31 ⇒ T32)))
  = (cod cs) ++ ((T22 ⟹[ l ] T32) ∷ [])

data PCastList : Type → Type → Type → Type → Set where
  [] : ∀ {T11 T12}
    → PCastList T11 T12 T11 T12
  _≪_ : ∀ {T11 T12 T21 T22 T31 T32}
    → PCastList T11 T12 T21 T22
    → L.Cast (‘ T21 ⊗ T22) (‘ T31 ⊗ T32)
    → PCastList T11 T12 T31 T32

view-cons : ∀ {T11 T12 T21 T22}
  → L.Value (‘ T11 ⊗ T12)
```

```
  → PCastList T11 T12 T21 T22
  → L.Value (‘ T21 ⊗ T22)
view-cons v [] = v
view-cons v (cs ≪ c) = view-cons v cs p⟨ c ⟩

lft : ∀ {T11 T12 T21 T22} → PCastList T11 T12 T21 T22 → CastList T11 T21
lft [] = []
lft (cs ≪ ((‘ T21 ⊗ T22) ⟹[ l ] (‘ T31 ⊗ T32)))
  = lft cs ++ ((T21 ⟹[ l ] T31) ∷ [])

rht : ∀ {T11 T12 T21 T22} → PCastList T11 T12 T21 T22 → CastList T12 T22
rht [] = []
rht (cs ≪ ((‘ T21 ⊗ T22) ⟹[ l ] (‘ T31 ⊗ T32)))
  = rht cs ++ ((T22 ⟹[ l ] T32) ∷ [])

data ErrorRelate {A B : Set} (A~B : A → B → Set) : Error Label A → Error Label B
  → Set where
  return : {a : A}{b : B} → (a~b : A~B a b) → ErrorRelate A~B (return a) (return b)
  raise : ∀ l → ErrorRelate A~B (raise l) (raise l)

mutual
  data EnvRelate : ∀ {Γ} → L.Env Γ → R.Env Γ → Set where
    []  : EnvRelate [] []
    _::_ : ∀ {Γ T}
      → {v : L.Value T}{u : R.Value T}
      → ValueRelate v u
      → {E : L.Env Γ}{F : R.Env Γ}
      → EnvRelate E F
```

```
    → EnvRelate (v :: E) (u :: F)


data ValueRelate : ∀ {T} → L.Value T → R.Value T → Set where
  dyn : ∀ {P}
    → (Pi : Injectable P)
    → {lv : L.Value (‘ P)}
    → {rv : R.Value (‘ P)}
    → ValueRelate lv rv
    ----------------
    → ValueRelate (dyn Pi lv) (dyn Pi rv)


  #t : ValueRelate #t #t
  #f : ValueRelate #f #f


  lam⟨_,_⇒_⟩ : ∀ {Γ T11 T12 T21 T22}
    → (lcs : FCastList T11 T12 T21 T22)
    → {rc1 : R.Cast T21 T11}
    → {rc2 : R.Cast T12 T22}
    → CastListRelate (dom lcs) rc1
    → CastListRelate (cod lcs) rc2
    → (e : Γ , T11  ⊢  T12)
    → {lE : L.Env Γ}
    → {rE : R.Env Γ}
    → (E : EnvRelate lE rE)
    ------
    → ValueRelate (view-lambda (lam e lE) lcs) (lam⟨ rc1 ⇒ rc2 ⟩ e rE)


  cons⟨_,_⊗_⟩ : ∀ {T1 T2 T3 T4}
```

```
      → (lcs : PCastList T1 T2 T3 T4)
      → {rc1 : R.Cast T1 T3}
      → {rc2 : R.Cast T2 T4}
      → CastListRelate (lft lcs) rc1
      → CastListRelate (rht lcs) rc2
      → {lv1 : L.Value T1}
      → {rv1 : R.Value T1}
      → (v1 : ValueRelate lv1 rv1)
      → {lv2 : L.Value T2}
      → {rv2 : R.Value T2}
      → (v2 : ValueRelate lv2 rv2)
      ----
      → ValueRelate (view-cons (cons lv1 lv2) lcs) (cons⟨ rc1 ⊗ rc2 ⟩ rv1 rv2)


lenv : ∀ {T}
  → {v : L.Env T}
  → {u : R.Env T}
  → EnvRelate v u
  → L.Env T
lenv {v = v} vr = v


renv : ∀ {T}
  → {v : L.Env T}
  → {u : R.Env T}
  → EnvRelate v u
  → R.Env T
renv {u = u} vr = u
```

```
lvalue : ∀ {T}
  → {v : L.Value T}
  → {u : R.Value T}
  → ValueRelate v u
  → L.Value T
lvalue {v = v} vr = v


rvalue : ∀ {T}
  → {v : L.Value T}
  → {u : R.Value T}
  → ValueRelate v u
  → R.Value T
rvalue {u = u} vr = u


lookup : ∀ {Γ T}
  → {lE : L.Env Γ}
  → {rE : R.Env Γ}
  → (E : EnvRelate lE rE)
  → (x : Γ ∋ T)
  → ValueRelate (L.lookup lE x) (R.lookup rE x)
lookup (c ∷ E) zero = c
lookup (c ∷ E) (suc x) = lookup E x


CastResultRelate : ∀ {T} → Error Label (L.Value T) → Error Label (R.Value T) → Set
CastResultRelate = ErrorRelate ValueRelate


data FrameRelate : ∀ {S T} → L.Frame S T → R.Frame S T → Set where
  app₁ : ∀ {Γ S T}
```

```
  → (e : Γ ⊢ S)
  → {lE : L.Env Γ}
  → {rE : R.Env Γ}
  → (E : EnvRelate lE rE)
  --------
  → FrameRelate {‘ S ⇒ T} {T} (app₁ e lE) (app₁ e rE)


app₂ : ∀ {S T}
  → {lv : L.Value (‘ S ⇒ T)}
  → {rv : R.Value (‘ S ⇒ T)}
  → (v : ValueRelate lv rv)
  --------
  → FrameRelate (app₂ lv) (app₂ rv)


if₁ : ∀ {Γ T}
  → (e2 : Γ ⊢ T)
  → (e3 : Γ ⊢ T)
  → {lE : L.Env Γ}
  → {rE : R.Env Γ}
  → (E : EnvRelate lE rE)
  ---------
  → FrameRelate (if₁ e2 e3 lE) (if₁ e2 e3 rE)


cons₁ : ∀ {Γ T1 T2}
  → (e2 : Γ ⊢ T2)
  → {lE : L.Env Γ}
  → {rE : R.Env Γ}
  → (E : EnvRelate lE rE)
```

```
    -----
    → FrameRelate {T1} (cons₁ e2 lE) (cons₁ e2 rE)


  cons₂ : ∀ {T1 T2}
    → {lv1 : L.Value T1}
    → {rv1 : R.Value T1}
    → (v1 : ValueRelate lv1 rv1)
    -----
    → FrameRelate {T2} (cons₂ lv1) (cons₂ rv1)


  car₁ : ∀ {T1 T2} → FrameRelate {‘ T1 ⊗ T2} {T1} car₁ car₁
  cdr₁ : ∀ {T1 T2} → FrameRelate {‘ T1 ⊗ T2} {T2} cdr₁ cdr₁


view-cont : ∀ {T1 T2 T3}
  → CastList T1 T2
  → L.Cont T2 T3
  → L.Cont T1 T3
view-cont [] k    = k
view-cont (c :: cs) k = [ □⟨ c ⟩ ] view-cont cs k


mutual
  data ContRelate : {T1 T2 : Type} → L.Cont T1 T2 → R.Cont T1 T2 → Set where
    [□⟨_⟩]_ : ∀ {T1 T2 T3 lc rc lk rk}
      → (c : CastListRelate {T1} {T2} lc rc)
      → (k : PreContRelate {T3} lk rk)
      ---
      → ContRelate (view-cont lc lk) (R.[□⟨ rc ⟩] rk)
```

```
  data PreContRelate {Z : Type} : ∀ {S} → L.Cont S Z → R.PreCont S Z → Set where
    □ : PreContRelate □ □

    [_] : ∀ {T1 T2}
      → {lF : L.Frame T1 T2}
      → {rF : R.Frame T1 T2}
      → (F : FrameRelate lF rF)
      → {lk : L.Cont T2 Z}
      → {rk : R.Cont T2 Z}
      → (k : ContRelate lk rk)
      ---
      → PreContRelate ([ lF ] lk) ([ rF ] rk)


lcont : ∀ {T1 T2}
  → {lk : L.Cont T1 T2}
  → {rk : R.Cont T1 T2}
  → (kk : ContRelate lk rk)
  ---
  → L.Cont T1 T2
lcont {lk = lk} kk = lk


rcont : ∀ {T1 T2}
  → {lk : L.Cont T1 T2}
  → {rk : R.Cont T1 T2}
  → (kk : ContRelate lk rk)
  ---
  → R.Cont T1 T2
rcont {rk = rk} kk = rk
```

```
lprecont : ∀ {T1 T2}
  → {lk : L.Cont T1 T2}
  → {rk : R.PreCont T1 T2}
  → (kk : PreContRelate lk rk)
  ---
  → L.Cont T1 T2
lprecont {lk = lk} kk = lk


rprecont : ∀ {T1 T2}
  → {lk : L.Cont T1 T2}
  → {rk : R.PreCont T1 T2}
  → (kk : PreContRelate lk rk)
  ---
  → R.PreCont T1 T2
rprecont {rk = rk} kk = rk


data OrdinaryStateRelate {Z : Type} : L.OrdinaryState Z → R.OrdinaryState Z
  → Set where
  expr : ∀ {Γ T1}
    → (e : Γ  ⊢  T1)
    → {lE : L.Env Γ}
    → {rE : R.Env Γ}
    → (E : EnvRelate lE rE)
    → {lK : L.Cont T1 Z}
    → {rK : R.Cont T1 Z}
    → (K : ContRelate lK rK)
    ------------
```

```
    → OrdinaryStateRelate (expr e lE lK) (expr e rE rK)


  cont : ∀ {T}
    → {lv : L.Value T}
    → {rv : R.Value T}
    → (v : ValueRelate lv rv)
    → {lK : L.Cont T Z}
    → {rK : R.Cont T Z}
    → (K : ContRelate lK rK)
    ------------
    → OrdinaryStateRelate (cont lv lK) (cont rv rK)


  halt : ∀ {lv rv}
    → (v : ValueRelate lv rv)
    → OrdinaryStateRelate (halt lv) (halt rv)


-- view-cont-cast : ∀ {T1 T2 T3}
--   → (v : L.Value T1)
--   → (cs : CastList T1 T2)
--   → (k : L.Cont T2 T3)
--   → (L.Progressing (return (cont v (view-cont cs k))))
-- view-cont-cast v [] k = cont v k
-- view-cont-cast v (c :: cs) k = cont v (L.[ □⟨ c ⟩ ] (view-cont cs k))


-- data ProgressingRelate {Z : Type} :
--   {lS : L.State Z}(lSp : L.Progressing lS)
--   {rS : R.State Z}(rSp : R.Progressing rS)
--   → Set
```

```
--   where
--   expr : ∀ {Γ T}
--     → (e : Γ ⊢ T)
--     → {lE : L.Env Γ}
--     → {rE : R.Env Γ}
--     → (E : EnvRelate lE rE)
--     → {lK : L.Cont T Z}
--     → {rK : R.Cont T Z}
--     → (K : ContRelate lK rK)
--     ------------
--     → ProgressingRelate (L.expr e lE lK) (R.expr e rE rK)


StateRelate : ∀ {T} → L.State T → R.State T → Set
StateRelate = ErrorRelate OrdinaryStateRelate


lstate : ∀ {T}
  → {ls : L.State T}
  → {rs : R.State T}
  → (ss : StateRelate ls rs)
  ---
  → L.State T
lstate {ls = ls} ss = ls


rstate : ∀ {T}
  → {ls : L.State T}
  → {rs : R.State T}
  → (ss : StateRelate ls rs)
  ---
```

```
  → R.State T

rstate {rs = rs} ss = rs
```

C.2 Main Bisimulation Proof

```
open import Types

open import R.BlameStrategies

open import S.CastADT

open import Bisimulation.ApplyCastBisimulate


module Bisimulation.BisimulationProof
  (Label : Set)
  (BS : BlameStrategy Label)
  (CADT : CastADT Label (BlameStrategy.Injectable BS))
  (lem-⌈_⌉ : the-apply-cast-lemma Label BS CADT)
  where


open BlameStrategy BS using (Injectable)

open import Variables using (∅)

open import Terms Label

open import Cast Label

open import Observables Label

open import Error using (Error; return; raise; _>=>_; _>>=_; >>=-return)


open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; sym)

open import Data.Sum using (_⊎_; inj₁; inj₂)

open import Data.Product using (_×_; _,_; ∃-syntax)


open import Bisimulation.Bisimulation using (_::_; [])
```

```
open import Bisimulation.BisimulationRelation Label BS CADT
  renaming (module L to L')


module L where
  open L' hiding (_++_; ⟦_⟧) public


  -- inteprete cast lists
  ⟦_⟧ : ∀ {T1 T2}
    → CastList T1 T2
    → (Value T1 → Error Label (Value T2))
  ⟦ [] ⟧ = return
  ⟦ x :: xs ⟧ = L'.⟦ x ⟧ >=> ⟦ xs ⟧


  lem-id : ∀ {T}
    → (v : Value T)
    → ⟦ [] ⟧ v ≡ return v
  lem-id v = refl


  lem-seq : ∀ {T1 T2 T3}
    → (xs : CastList T1 T2)
    → (ys : CastList T2 T3)
    → (∀ v → ⟦ xs ++ ys ⟧ v ≡ (⟦ xs ⟧ >=> ⟦ ys ⟧) v)
  lem-seq [] ys v = refl
  lem-seq (x :: xs) ys v with L'.⟦ x ⟧ v
  ... | return v' = lem-seq xs ys v'
  ... | raise l = refl


lem-⟦_⟧ : ∀ {T1 T2 lv rv lc rc}
```

```
  → CastListRelate {T1} {T2} lc rc
  → ValueRelate {T1} lv rv
  → CastResultRelate (L.⟦ lc ⟧ lv) (R.⟦ rc ⟧ rv)
lem-⟦ id ⟧ v
  rewrite R.lem-id _ (rvalue v)
    = return v
lem-⟦ just c ⟧ v
  rewrite >>=-return (L'.⟦ c ⟧ (lvalue v))
    = lem-⌈ c ⌉ v
lem-⟦ c1 ⨟ c2 ⟧ v
  rewrite L.lem-seq (lcast c1) (lcast c2) (lvalue v)
        | R.lem-seq (rcast c1) (rcast c2) (rvalue v)
  with L.⟦ lcast c1 ⟧ (lvalue v) | R.⟦ rcast c1 ⟧ (rvalue v) | lem-⟦ c1 ⟧ v
... | .(return _) | .(return _) | return v' = lem-⟦ c2 ⟧ v'
... | .(raise _)  | .(raise _)  | raise l   = raise l

open L using (error; halt; cont; expr; it)
open R using (error; halt; cont; expr; it; [□⟨_⟩]_)

load : ∀ {T} → (e : ∅ ⊢ T) → StateRelate (L.load e) (R.load e)
load e = return (expr e [] ([□⟨ id ⟩] □))

observe : ∀ {T}
  → {lv : L.Value T}
  → {rv : R.Value T}
  → ValueRelate lv rv
  → L.observe lv ≡ R.observe rv
observe (dyn Pi v) = refl
```

```
observe #t = refl

observe #f = refl

observe (lam⟨ [] , c ⇒ d ⟩ e E) = refl

observe (lam⟨ lcs ≪ lc , c ⇒ d ⟩ e E) = refl

observe (cons⟨ [] , c ⊗ d ⟩ v1 v2) = refl

observe (cons⟨ lcs ≪ lc , c ⊗ d ⟩ v1 v2) = refl


view-cont-++ : ∀ {T1 T2 T3 Z}
  → (cs : CastList T1 T2)
  → (ds : CastList T2 T3)
  → (k  : L.Cont T3 Z)
  → view-cont cs (view-cont ds k) ≡ view-cont (cs ++ ds) k
view-cont-++ []       ds k = refl
view-cont-++ (c ∷ cs) ds k rewrite view-cont-++ cs ds k = refl


-- the next two lemmas are not useful yet in formal proof
lemma-mk-cont : ∀ {T1 T2}
  → {lk : L.Cont T1 T2}
  → {rk : R.PreCont T1 T2}
  → PreContRelate lk rk
  → ContRelate lk (R.mk-cont rk)
lemma-mk-cont k = [□⟨ id ⟩] k


lemma-ext-cont : ∀ {T1 T2 T3}
  → {lc : CastList T1 T2}
  → {rc : R.Cast T1 T2}
  → CastListRelate lc rc
  → {lk : L.Cont T2 T3}
```

```
  → {rk : R.Cont T2 T3}
  → ContRelate lk rk
  → ContRelate (view-cont lc lk) (R.ext-cont rc rk)
lemma-ext-cont c ([□⟨ d ⟩] k)
  rewrite view-cont-++ (lcast c) (lcast d) (lprecont k)
  = [□⟨ c ⨟ d ⟩] k


lem->>= : ∀ {T1 T2 lr rr}
  → CastResultRelate {T1} lr rr
  → {lf : L.Value T1 → L.State T2}
  → {rf : R.Value T1 → R.State T2}
  → (f : ∀ {lv rv} → ValueRelate {T1} lv rv → StateRelate (lf lv) (rf rv))
  → StateRelate (lr >>= lf) (rr >>= rf)
lem->>= (return v) f = f v
lem->>= (raise l)  f = raise l


done : ∀ {T}
  → {lS : L.State T}
  → {rS : R.State T}
  → StateRelate lS rS
  → ∃[ lS' ]
    ∃[ rS' ]
      (lS L.⟶* lS' ×
       rS R.⟶* rS' ×
       StateRelate lS' rS')
done S = _ , _ , [] , [] , S


step : ∀ {T}
```

```
  → {lS lS' : L.State T}
  → {rS rS' : R.State T}
  → (lS L.⟶* lS')
  → (rS R.⟶* rS')
  → ∃[ lS'' ]
    ∃[ rS'' ]
      (lS' L.⟶* lS'' ×
       rS' R.⟶* rS'' ×
       StateRelate lS'' rS'')
  → ∃[ lS'' ]
    ∃[ rS'' ]
      (lS L.⟶* lS'' ×
       rS R.⟶* rS'' ×
       StateRelate lS'' rS'')
step lxs rxs (_ , _ , lys , rys , S) = _ , _ , (lxs L'.++ lys) , (rxs R.++ rys) , S


cnd : ∀ {A lv rv} → ValueRelate {‘ B} lv rv → (x y : A)
  → L.cnd lv x y ≡ R.cnd rv x y
cnd #t x y = refl
cnd #f x y = refl


lem-L-apply-cont : ∀ {T1 T2 T3}
  → (v : L.Value T1)
  → (cs : CastList T1 T2)
  → (k : L.Cont T2 T3)
  → (L.apply-cont v (view-cont cs k))
    L.⟶*
    (L.⟦ cs ⟧ v >>= λ v' → (L.apply-cont v' k))
```

```
lem-L-apply-cont v [] k = []

lem-L-apply-cont v (c ∷ cs) k with L'.⟦ c ⟧ v

... | return v' = it (cont v' (view-cont cs k)) ∷ lem-L-apply-cont v' cs k

... | raise l = []


++-assoc : ∀ {T1 T2 T3 T4}

  → (xs : CastList T1 T2)

  → (ys : CastList T2 T3)

  → (zs : CastList T3 T4)

  → (xs ++ (ys ++ zs)) ≡ ((xs ++ ys) ++ zs)

++-assoc []       ys zs = refl

++-assoc (x ∷ xs) ys zs rewrite ++-assoc xs ys zs = refl


lemma-L-do-app : ∀ {S1 T1 S2 T2 T}

  → (f  : L.Value (‘ S1 ⇒ T1))

  → (cs : FCastList S1 T1 S2 T2)

  → (a  : L.Value S2)

  → (k  : L.Cont T2 T)

  → (L.do-app (view-lambda f cs) a k)

      L.⟶*

    (L.⟦ dom cs ⟧ a >>= λ v → (L.do-app f v (view-cont (cod cs) k)))

lemma-L-do-app f []       a k = L.[]

lemma-L-do-app f (cs ≪ (‘ S1 ⇒ T1 ⟹[ l ] ‘ S2 ⇒ T2)) a k

  = it (cont _ _) ∷ next

  where

  next : (L'.⟦ S2 ⟹[ l ] S1 ⟧ a

          >>= λ v →

          (return (L.cont v (L.[ L.app₂ (view-lambda f cs) ]
```

```
                              L.[ L.□⟨ T1 ⟹[ l ] T2 ⟩ ] k))))

          L.⟶*

         (L'.⟦ S2 ⟹[ l ] S1 ⟧ a

          >>= L.⟦ dom cs ⟧

          >>= λ v →

          L.do-app f v (view-cont (cod cs ++ ((T1 ⟹[ l ] T2) ∷ [])) k))

  next with L'.⟦ S2 ⟹[ l ] S1 ⟧ a

  next | raise l  = L.[]

  next | return v

    rewrite sym (view-cont-++ (cod cs) ((T1 ⟹[ l ] T2) ∷ []) k)

    = it (cont _ _) ∷ lemma-L-do-app f cs v (L.[ L.□⟨ T1 ⟹[ l ] T2 ⟩ ] k)


do-app : ∀ {T1 T2 Z lv1 rv1 lv2 rv2 lk rk}

  → ValueRelate {‘ T1 ⇒ T2} lv1 rv1

  → ValueRelate {T1} lv2 rv2

  → ContRelate lk rk

  → ∃[ lS' ]

    ∃[ rS' ]

      (L.do-app lv1 lv2 lk L.⟶* lS' ×

       R.do-app rv1 rv2 rk R.⟶* rS' ×

       StateRelate {Z} lS' rS')

do-app (lam⟨ lcs , c1 ⇒ c2 ⟩ e E) a k

  = step (lemma-L-do-app (L.lam e (lenv E)) lcs (lvalue a) (lcont k))

         []

         (done (lem->>= (lem-⟦ c1 ⟧ a)

                        λ v → return (expr e (v ∷ E) (lemma-ext-cont c2 k))))


lem-L-do-car : ∀ {T1 T2 T3 T4 Z}
```

```
  → (v1 : L.Value T1)
  → (v2 : L.Value T2)
  → (cs : PCastList T1 T2 T3 T4)
  → (k : L.Cont T3 Z)
  → (L.do-car (view-cons (L.cons v1 v2) cs) k)
      L.⟶*
    (L.⟦ lft cs ⟧ v1 >>= λ v' → (return (cont v' k)))
lem-L-do-car v1 v2 []       k = []
lem-L-do-car v1 v2 (cs ≪ ((‘ T1 ⊗ T2) ⟹[ l ] (‘ T3 ⊗ T4))) k
  with lem-L-do-car v1 v2 cs (view-cont ((T1 ⟹[ l ] T3) ∷ []) k)
... | IH
  rewrite L.lem-seq (lft cs) ((T1 ⟹[ l ] T3) ∷ []) v1
  = it (cont _ _) ∷ (IH L'.++ next)
  where
    next : (L.⟦ lft cs ⟧ v1 >>=
       (λ v' → return (cont v' (L.[ L.□⟨ T1 L.⟹[ l ] T3 ⟩ ] k))))
      L.⟶*
      ((L.⟦ lft cs ⟧ v1 >>= (λ x → L'.⟦ T1 L.⟹[ l ] T3 ⟧ x >>= return))
       >>= (λ v' → return (cont v' k)))
    next
      with L.⟦ lft cs ⟧ v1
    ... | raise l' = []
    ... | return v1'
      rewrite >>=-return (L'.⟦ T1 ⟹[ l ] T3 ⟧ v1')
      = it (cont _ _) ∷ []


do-car : ∀ {T1 T2 Z lv rv lk rk}
  → ValueRelate {‘ T1 ⊗ T2} lv rv
```

```
  → ContRelate lk rk
  → ∃[ lS' ]
    ∃[ rS' ]
      (L.do-car lv lk L.⟶* lS' ×
       R.do-car rv rk R.⟶* rS' ×
       StateRelate {Z} lS' rS')
do-car (cons⟨ lcs , c1 ⊗ c2 ⟩ v1 v2) k
  = step (lem-L-do-car (lvalue v1) (lvalue v2) lcs (lcont k)) []
         (done (lem->>= (lem-⟦ c1 ⟧ v1) λ v1' → return (cont v1' k)))


lem-L-do-cdr : ∀ {T1 T2 T3 T4 Z}
  → (v1 : L.Value T1)
  → (v2 : L.Value T2)
  → (cs : PCastList T1 T2 T3 T4)
  → (k : L.Cont T4 Z)
  → (L.do-cdr (view-cons (L.cons v1 v2) cs) k)
      L.⟶*
    (L.⟦ rht cs ⟧ v2 >>= λ v' → (return (cont v' k)))
lem-L-do-cdr v1 v2 []       k = []
lem-L-do-cdr v1 v2 (cs ≪ ((‘ T1 ⊗ T2) ⟹[ l ] (‘ T3 ⊗ T4))) k
  with lem-L-do-cdr v1 v2 cs (view-cont ((T2 ⟹[ l ] T4) ∷ []) k)
... | IH
  rewrite L.lem-seq (rht cs) ((T2 ⟹[ l ] T4) ∷ []) v2
  = it (cont _ _) ∷ (IH L'.++ next)
  where
    next : (L.⟦ rht cs ⟧ v2 >>=
       (λ v' → return (cont v' (L.[ L.□⟨ T2 ⟹[ l ] T4 ⟩ ] k))))
      L.⟶*
```

```
      ((L.⟦ rht cs ⟧ v2 >>= (λ x → L'.⟦ T2 ⟹[ l ] T4 ⟧ x >>= return))
       >>= (λ v' → return (cont v' k)))
    next
      with L.⟦ rht cs ⟧ v2
    ... | raise l' = []
    ... | return v'
      rewrite >>=-return (L'.⟦ T2 ⟹[ l ] T4 ⟧ v')
      = it (cont _ _) ∷ []


do-cdr : ∀ {T1 T2 Z lv rv lk rk}
  → ValueRelate {‘ T1 ⊗ T2} lv rv
  → ContRelate lk rk
  → ∃[ lS' ]
    ∃[ rS' ]
      (L.do-cdr lv lk L.⟶* lS' ×
       R.do-cdr rv rk R.⟶* rS' ×
       StateRelate {Z} lS' rS')
do-cdr (cons⟨ lcs , c1 ⊗ c2 ⟩ v1 v2) k
  = step (lem-L-do-cdr (lvalue v1) (lvalue v2) lcs (lcont k)) []
         (done (lem->>= (lem-⟦ c2 ⟧ v2) λ v' → return (cont v' k)))


apply-cont : ∀ {T1 T2 lv rv lk rk}
  → ValueRelate {T1} lv rv
  → PreContRelate {T2} lk rk
  → ∃[ lS' ]
    ∃[ rS' ]
      (L.apply-cont lv lk L.⟶* lS' ×
       R.apply-cont rv rk R.⟶* rS' ×
```

```
       StateRelate lS' rS')
apply-cont v □ = done (return (halt v))
apply-cont v ([ app₁ e E ] k) = done (return (expr e E ([□⟨ id ⟩] [ app₂ v ] k)))
apply-cont v ([ app₂ v1 ] k)  = do-app v1 v k
apply-cont v ([ if₁ e2 e3 E ] k)
  with L.cnd (lvalue v) e2 e3 | R.cnd (rvalue v) e2 e3 | cnd v e2 e3
... | e | e | refl = done (return (expr e E k))
apply-cont v ([ cons₁ e2 E ] k)
  = done (return (expr e2 E ([□⟨ id ⟩] [ cons₂ v ] k)))
apply-cont v ([ cons₂ v1 ] k)
  = done (return (cont (cons⟨ [] , id ⊗ id ⟩ v1 v) k))
apply-cont v ([ car₁ ] k) = do-car v k
apply-cont v ([ cdr₁ ] k) = do-cdr v k


progress : ∀ {T}
  → {lS : L.State T}(lSp : L.Progressing lS)
  → {rS : R.State T}(rSp : R.Progressing rS)
  → StateRelate lS rS
  → ∃[ lS' ]
    ∃[ rS' ]
      (L.progress lSp L.⟶* lS' ×
       R.progress rSp R.⟶* rS' ×
       StateRelate lS' rS')
progress (expr (var x) lE lK)
         (expr (var x) rE rK)
         (return (expr (var x) E K))
         = done (return (cont (lookup E x) K))
progress (expr #t lE lK)
```

```
         (expr #t rE rK)
         (return (expr #t E K))
         = done (return (cont #t K))
progress (expr #f lE lK)
         (expr #f rE rK)
         (return (expr #f E K))
         = done (return (cont #f K))
progress (expr (if e e₁ e₂) lE lK)
         (expr (if e e₁ e₂) rE rK)
         (return (expr (if e e₁ e₂) E K))
         = done (return (expr e E (lemma-mk-cont ([ if₁ e₁ e₂ E ] K))))
progress (expr (lam e) lE lK)
         (expr (lam e) rE rK)
         (return (expr (lam e) E K))
         = done (return (cont (lam⟨ [] , id ⇒ id ⟩ e E) K))
progress (expr (app e e₁) lE lK)
         (expr (app e e₁) rE rK)
         (return (expr .(app e e₁) E K))
         = done (return (expr e E (lemma-mk-cont ([ app₁ e₁ E ] K))))
progress (expr (cons e e₁) lE lK)
         (expr (cons e e₁) rE rK)
         (return (expr .(cons e e₁) E K))
         = done (return (expr e E ((lemma-mk-cont ([ cons₁ e₁ E ] K)))))
progress (expr (e ⟨ c ⟩) lE lK)
         (expr (e ⟨ c ⟩) rE rK)
         (return (expr (e ⟨ c ⟩) E K))
         = done (return (expr e E (lemma-ext-cont (just c) K)))
progress (expr (car e) lE lK)
```

```
         (expr (car e) rE rK)
         (return (expr (car e) E K))
         = done (return (expr e E (lemma-mk-cont ([ car₁ ] K))))
progress (expr (cdr e) lE lK)
         (expr (cdr e) rE rK)
         (return (expr (cdr e) E K))
         = done (return (expr e E (lemma-mk-cont ([ cdr₁ ] K))))
-- progress (expr (blame l) lE lK)
--          (expr (blame l) rE rK)
--          (return (expr (blame l) E K))
--          = done (raise l)
progress (cont lv lk)
         (cont rv rk)
         (return (cont v k))
  with lk | rk | k
... | .(view-cont (lcast c) (lprecont k'))
    | .([□⟨ rcast c ⟩] rprecont k')
    | [□⟨ c ⟩] k'
  with lem-L-apply-cont lv (lcast c) (lprecont k')
... | lSâ§ijlS'
  with L.⟦ (lcast c) ⟧ (lvalue v) | R.⟦ (rcast c) ⟧ (rvalue v)
    | lem-⟦ c ⟧ v
... | .(raise _)  | .(raise _)  | raise l   = step lSâ§ijlS' [] (done (raise l))
... | .(return _) | .(return _) | return v' = step lSâ§ijlS' [] (apply-cont v' k')


final-or-progressing : ∀ {T}
  → {ls : L.State T}
  → {rs : R.State T}
```

```
  → StateRelate ls rs
  → (L.Final ls × R.Final rs)
    ⊎
    (L.Progressing ls × R.Progressing rs)
final-or-progressing (return (expr e E K))
  = inj₂ (expr e (lenv E) (lcont K) , expr e (renv E) (rcont K))
final-or-progressing (return (cont v K))
  = inj₂ (cont (lvalue v) (lcont K) , cont (rvalue v) (rcont K))
final-or-progressing (return (halt v))
  = inj₁ (halt (lvalue v) , halt (rvalue v))
final-or-progressing (raise l)
  = inj₁ (error l , error l)


-- the main lemma
safety : ∀ {T}
  → {ls : L.State T}
  → {rs : R.State T}
  → StateRelate ls rs
  → (L.Final ls × R.Final rs)
    ⊎
    (∃[ ls' ]
     ∃[ rs' ]
       (ls L.⟶+ ls' ×
        rs R.⟶+ rs' ×
        StateRelate ls' rs'))
safety S with final-or-progressing S
safety S | inj₁ (lSf , rSf) = inj₁ (lSf , rSf)
safety S | inj₂ (lSp , rSp) with progress lSp rSp S
```

```
safety S | inj₂ (lSp , rSp) | lS'' , rS'' , lS'â§ijlS'' , rS'â§ijrS'' , S''
  = inj₂ (lS'' , rS'' , (it lSp :: lS'â§ijlS'') , (it rSp :: rS'â§ijrS'') , S'')


module Foo {T : Type} where
  import Bisimulation.Bisimulation
  module CorrectnessTheorems =
    Bisimulation.Bisimulation.Theorems (L.system {T = T}) R.system StateRelate safety
  open CorrectnessTheorems using (thm-final-LR; thm-final-RL) public


correctness-l : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → L.Evalo e o
  ---
  → R.Evalo e o
correctness-l {e = e} (L.val xs) with Foo.thm-final-LR (load e) xs (halt _)
correctness-l {e = e} (L.val xs) | _ , ys , (R.halt rv) , return (halt v)
  rewrite observe v
  = R.val ys
correctness-l {e = e} (L.err xs) with Foo.thm-final-LR (load e) xs (error _)
correctness-l {e = e} (L.err xs) | _ , ys , (R.error l) , raise l
  = R.err ys


correctness-r : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → R.Evalo e o
  ---
```

```
  → L.Evalo e o
correctness-r {e = e} (R.val xs) with Foo.thm-final-RL (load e) xs (halt _)
correctness-r {e = e} (R.val xs) | _ , ys , (L.halt rv) , return (halt v)
  rewrite sym (observe v)
  = L.val ys
correctness-r {e = e} (R.err xs) with Foo.thm-final-RL (load e) xs (error _)
correctness-r {e = e} (R.err xs) | _ , ys , (L.error l) , raise l
  = L.err ys
```

C.3 Cast Application Should Preserves Bisimulation Relation

```
open import R.BlameStrategies
open import S.CastADT

module Bisimulation.ApplyCastBisimulate
  (Label : Set)
  (BS : BlameStrategy Label)
  (CADT : CastADT Label (BlameStrategy.Injectable BS))
  where

open import Bisimulation.BisimulationRelation Label BS CADT
  using (module L; module R; ValueRelate; CastResultRelate)

open import Cast Label using (Cast)
open BlameStrategy BS using (Injectable) public

the-apply-cast-lemma : Set
the-apply-cast-lemma = ∀ {S T lv rv}
    → (c : Cast S T)
```

```
    → ValueRelate {S} lv rv
    → CastResultRelate (L.⟦ c ⟧ lv) (R.⟦ R.⌈ c ⌉ ⟧ rv)
```

C.4 Lazy D Cast Application Preserves Bisimulation Relation

```
open import R.BlameStrategies using (BlameStrategy; LazyDBS)
open import S.CastADT
open import S.LazyDCastADT using (LazyD)

module Bisimulation.LazyDApplyCast
  (Label : Set)
  (CADT : CastADT Label (BlameStrategy.Injectable (LazyDBS Label)))
  (CADTLazyD : LazyD Label CADT)
  where

open import Types
open import Bisimulation.BisimulationRelation Label (LazyDBS Label) CADT
open import Cast Label using (Cast; _⟹[_]_)
open LazyD CADTLazyD

open import Relation.Nullary using (yes; no)
open import Data.Unit using (âŁd'; tt)

lem-⟦_⟧' : ∀ {P Q lv rv}
    → (c : Cast (‘ P) (‘ Q))
    → ValueRelate lv rv
    → CastResultRelate (L.⟦ c ⟧ lv)
                       (R.⟦ R.⌈ c ⌉ ⟧ rv)
lem-⟦_⟧' (‘ P ⟹[ l ] ‘ Q) v with (‘ P) ⌣? (‘ Q)
```

```
lem-⟦_⟧' (‘ B ⟹[ l ] ‘ B) v | yes ⌣B rewrite eq-B l (rvalue v) = return v

lem-⟦_⟧' (‘ T11 ⇒ T12 ⟹[ l ] ‘ T21 ⇒ T22) (lam⟨ lcs , c1 ⇒ c2 ⟩ e E) | yes ⌣⇒

  rewrite eq-⇒ T21 T22 T11 T12 l (rcast c1) (rcast c2) e (renv E)

  = return (lam⟨ lcs ≪ _

               , just (T21 ⟹[ l ] T11) ⨟ c1 ⇒ c2 ⨟ just (T12 ⟹[ l ] T22) ⟩

               e E)

lem-⟦_⟧' (‘ T11 ⊗ T12 ⟹[ l ] ‘ T21 ⊗ T22) (cons⟨ lcs , c1 ⊗ c2 ⟩ v1 v2) | yes ⌣⊗

  rewrite eq-⊗ T21 T22 T11 T12 l (rcast c1) (rcast c2) (rvalue v1) (rvalue v2)

  = return (cons⟨ lcs ≪ _

                , c1 ⨟ just (T11 ⟹[ l ] T21) ⊗ c2 ⨟ just (T12 ⟹[ l ] T22) ⟩

                v1 v2)

lem-⟦_⟧' (.(‘ _) ⟹[ l ] .(‘ _)) v | no ¬p

  rewrite eq-¬⌣ l ¬p (rvalue v)

  = raise l


lem-⟦_⟧ : ∀ {S T lv rv}

    → (c : Cast S T)

    → ValueRelate lv rv

    → CastResultRelate (L.⟦ c ⟧ lv)

                       (R.⟦ R.⌈ c ⌉ ⟧ rv)

lem-⟦_⟧ (∗   ⟹[ l ] ∗)   v rewrite eq-∗∗ l (rvalue v) = return v

lem-⟦_⟧ (‘ P ⟹[ l ] ∗)   v rewrite eq-P∗ l (rvalue v) = return (dyn (same P) v)

lem-⟦_⟧ (‘ P ⟹[ l ] ‘ Q) v = lem-⟦_⟧' (‘ P ⟹[ l ] ‘ Q) v

lem-⟦_⟧ (∗   ⟹[ l ] ‘ Q) (dyn (same P) v)

  rewrite eq-∗P Q P l (rvalue v)

  = lem-⟦_⟧' (‘ P ⟹[ l ] ‘ Q) v
```

C.5 Lazy UD Cast Application Preserves Bisimulation Relation

```
open import R.BlameStrategies using (BlameStrategy; LazyUDBS)
open import S.CastADT
open import S.LazyUDCastADT using (LazyUD)


module Bisimulation.LazyUDApplyCast
  (Label : Set)
  (CADT : CastADT Label (BlameStrategy.Injectable (LazyUDBS Label)))
  (CADTLazyUD : LazyUD Label CADT)
  where


open import Types
open import Bisimulation.BisimulationRelation Label (LazyUDBS Label) CADT
open import Error
open import Cast Label using (Cast; _⟹[_]_)
open LazyUD CADTLazyUD


open import Relation.Nullary using (yes; no)
open import Data.Unit using (âŁd'; tt)
open import Data.Product using (Îč; _×_ ; ∃-syntax; _,_)
open import Relation.Binary.PropositionalEquality using (_≡_; refl)
open import Data.Empty using (⊥; ⊥-elim)


open R.BlameStrategies.LazyUD Label using (project)


lem-project : ∀ {Q lv rv}
  → (l : Label)
  → (gQ : Ground Q)
  → ValueRelate lv rv
```

```
  → CastResultRelate (project lv l gQ)
                     (R.⟦ R.⌈ ∗ ⟹[ l ] ‘ Q ⌉ ⟧ rv)
lem-project {Q = Q} l gQ (dyn gP v) with (‘ unground gP) =? (‘ unground gQ)
... | yes refl
  rewrite eq-∗I-succ (rvalue v) l gP
  = return v
... | no ¬P≡Q
  rewrite eq-∗I-fail (rvalue v) l gP gQ ¬P≡Q
  = raise l


lem-proxy : ∀ {P Q}
    → {lv : L.Value (‘ P)}
    → {rv : R.Value (‘ P)}
    → ValueRelate lv rv
    → (c : Cast (‘ P) (‘ Q))
    → (p : (‘ P) ⌣ (‘ Q))
    → ∃[ ru ](
         (R.⟦ R.⌈ c ⌉ ⟧ rv ≡ return ru) ×
         ValueRelate (L.add-proxy lv c p) ru
      )
lem-proxy v (‘ B ⟹[ l ] ‘ B) ⌣B rewrite eq-B l (rvalue v) = _ , refl , v
lem-proxy (lam⟨ lcs , c1 ⇒ c2 ⟩ e E) (‘ S1 ⇒ T1 ⟹[ l ] ‘ S2 ⇒ T2) ⌣⇒
  rewrite eq-⇒ S2 T2 S1 T1 l (rcast c1) (rcast c2) e (renv E)
  = _ , refl
  , lam⟨ lcs ≪ _ , just (S2 ⟹[ l ] S1) ⨟ c1 ⇒ c2 ⨟ just (T1 ⟹[ l ] T2) ⟩ e E
lem-proxy (cons⟨ lcs , c1 ⊗ c2 ⟩ v1 v2) (‘ L1 ⊗ R1 ⟹[ l ] ‘ L2 ⊗ R2) ⌣⊗
  rewrite eq-⊗ L2 R2 L1 R1 l (rcast c1) (rcast c2) (rvalue v1) (rvalue v2)
  = _ , refl
```

```
  , cons⟨ lcs ≪ _ , c1 ⨟ just (L1 ⟹[ l ] L2) ⊗ c2 ⨟ just (R1 ⟹[ l ] R2) ⟩ v1 v2


lem-⋆P : ∀ {P lv rv}
  → (c : Cast ⋆ (‘ P))
  → ValueRelate lv rv
  → CastResultRelate (L.⟦ c ⟧ lv)
                     (R.⟦ R.⌈ c ⌉ ⟧ rv)
lem-⋆P (⋆ ⟹[ l ] ‘ Q) v
  with ground? Q
... | yes gQ = lem-project l gQ v
... | no ¬gQ
  rewrite eq-⋆P (rvalue v) l ¬gQ
  with project (lvalue v) l (ground-Ground Q)
     | R.⟦ R.⌈ ⋆ ⟹[ l ] (‘ ground Q) ⌉ ⟧ (rvalue v)
     | lem-project l (ground-Ground Q) v
... | .(raise  _) | .(raise  _) | raise l’  = raise l’
... | .(return _) | .(return _) | return v’
  with lem-proxy v’ ((‘ ground Q) ⟹[ l ] (‘ Q)) (⌣sym (ground-⌣ Q))
... | _ , eq , v’’ rewrite eq = return v’’


lem-P⋆ : ∀ {P lv rv}
  → (c : Cast (‘ P) ⋆)
  → ValueRelate lv rv
  → CastResultRelate (L.⟦ c ⟧ lv)
                     (R.⟦ R.⌈ c ⌉ ⟧ rv)
lem-P⋆ (‘ P ⟹[ l ] ⋆) v with ground? P
... | yes gP rewrite eq-I⋆ (rvalue v) l gP = return (dyn gP v)
... | no ¬gP rewrite eq-P⋆ (rvalue v) l ¬gP
```

```
  with lem-proxy v (‘ P ⟹[ l ] ‘ ground P) (ground-⌣ P)
... | _ , eq , v’
  rewrite eq | eq-I⋆ (rvalue v’) l (ground-Ground P)
  = return (dyn (ground-Ground P) v’)

lem-⟦_⟧ : ∀ {S T lv rv}
    → (c : Cast S T)
    → ValueRelate lv rv
    → CastResultRelate (L.⟦ c ⟧ lv)
                       (R.⟦ R.⌈ c ⌉ ⟧ rv)
lem-⟦ ⋆   ⟹[ l ] ⋆   ⟧ v rewrite eq-⋆⋆ l (rvalue v) = return v
lem-⟦ ⋆   ⟹[ l ] ‘ Q ⟧ v = lem-⋆P (⋆ ⟹[ l ] ‘ Q) v
lem-⟦ ‘ P ⟹[ l ] ⋆   ⟧ v = lem-P⋆ (‘ P ⟹[ l ] ⋆) v
lem-⟦ ‘ P ⟹[ l ] ‘ Q ⟧ v with (‘ P) ⌣? (‘ Q)
... | no ¬P⌣Q rewrite eq-¬⌣ (rvalue v) l ¬P⌣Q = raise l
... | yes P⌣Q with lem-proxy v ((‘ P) ⟹[ l ] (‘ Q)) P⌣Q
... | _ , eq , v’ rewrite eq = return v’
```

C.6 Other Helper Definitions

```
module Bisimulation.Bisimulation
  where

open import Data.Empty using (⊥; ⊥-elim)
open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Data.Sum using (_⊎_ ; inj₁; inj₂)
open import Data.Product using (Îč; _×_ ; Îč-syntax; ∃-syntax; _,_; proj₁; proj₂)
open import Data.Nat using (ℕ; zero; suc)
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; sym; subst)
```

```
-- This is a parameterized bisimulation proof between any two systems.


-- A *system* is a deterministic transition system, which includes
--   - a set of states
--   - a transition relation
--   - a theorem saying that final state cannot transition
--   - a theorem saying that the transition relation is deterministic


-- The parameterized bisimulation proof claims that
-- given a bisimulation relation _âL'Ĺ_
-- If s₁ âL'Ĺ s₂ implies that either
--   - s₁ final and s₂ final, or
--   - there exist s₃ and s₄ such that s₁ ⟶+ s₃ and s₂ ⟶+ s₄ and s₃ âL'Ĺ s₄
-- then for all s₁ âL'Ĺ s₂
--   - s₁ ⟶* s₃ and s₃ final implies
--     s₂ ⟶* s₄ and s₄ final and s₃ âL'Ĺ s₄ for some s₄
--   - s₂ ⟶* s₄ and s₄ final implies
--     s₁ ⟶* s₃ and s₃ final and s₃ âL'Ĺ s₄ for some s₃


record System (State : Set) : Set₁ where
  field
    _⟶_ : State → State → Set
    Final : State → Set
    final-progressing-absurd : ∀ {s t}
      → Final s
      → (s ⟶ t)
      → ⊥
```

```
  deterministic : ∀ {s t1 t2}
    → s ⟶ t1
    → s ⟶ t2
    → t1 ≡ t2


data _⟶*_ : State → State → Set where
  [] : {s : State}
    → s ⟶* s


  _::_ : ∀ {s1 s2 s3}
    → (x  : s1 ⟶ s2)
    → (xs : s2 ⟶* s3)
    → s1 ⟶* s3


record _⟶+_ (s s' : State) : Set where
  constructor _::_
  field
    {t} : State
    x  : s ⟶  t
    xs : t ⟶* s'


data Prefix : {s1 s2 s3 : State} → s1 ⟶* s2 → s1 ⟶* s3 → Set where
  zero : ∀ {s1 s3}
    → {xs : s1 ⟶* s3}
    → Prefix [] xs


  suc : ∀ {s0 s1 s2 s3}
    → {x : s0 ⟶ s1}
```

```
      → {y : s0 ⟶ s1}
      → {xs : s1 ⟶* s2}
      → {ys : s1 ⟶* s3}
      → Prefix xs ys
      → Prefix (x ∷ xs) (y ∷ ys)

  _++_ : ∀ {s1 s2 s3} → s1 ⟶* s2 → s2 ⟶* s3 → s1 ⟶* s3
  [] ++ ys = ys
  (x ∷ xs) ++ ys = x ∷ (xs ++ ys)

open System using ([]; _∷_) public

module Lemmas
  {lState rState : Set}
  (lSystem : System lState)
  (rSystem : System rState)
  (_~_ : lState → rState → Set)
  (safety : {ls : lState}{rs : rState}
    → ls ~ rs
    → (System.Final lSystem ls × System.Final rSystem rs)
      ⊎
      ∃[ ls' ]
      ∃[ rs' ](
        System._⟶+_ lSystem ls ls' ×
        System._⟶+_ rSystem rs rs' ×
        (ls' ~ rs')))
  where
```

```
open System using ([]; _::_; zero; suc)


module L where
  open System lSystem public


module R where
  open System rSystem public


prefix : ∀ {ls}
  → ∀ {ls'}
  → (xs : ls L.⟶* ls')
  → L.Final ls'
  → ∀ {ls*}
  → (ys : ls L.⟶* ls*)
  → L.Prefix ys xs
prefix xs halt [] = zero
prefix [] halt (y :: ys) = ⊥-elim (L.final-progressing-absurd halt y)
prefix (x :: xs) halt (y :: ys) with L.deterministic x y
... | refl = suc (prefix xs halt ys)


mutual
  -- the job of this helper is cutting prefix
  lem-final-helper : ∀ {ls1 ls2 rs}
    → (ss : ls2 ~ rs)
    → ∀ {ls'}
    → (xs : ls1 L.⟶* ls2)
    → (ys : ls1 L.⟶* ls')
    → L.Prefix xs ys
```

```
  → L.Final ls'
  → ∃[ rs' ](
       rs R.⟶* rs' ×
       R.Final rs' ×
       ls' ~ rs'
    )
lem-final-helper ss [] ys zero ht = lem-final ss ys ht
lem-final-helper ss (x :: xs) (y :: ys) (suc n) ht = lem-final-helper ss xs ys n ht


lem-final : ∀ {ls rs}
  → (ss : ls ~ rs)
  → ∀ {ls'}
  → ls L.⟶* ls'
  → L.Final ls'
  → ∃[ rs' ](
       rs R.⟶* rs' ×
       R.Final rs' ×
       ls' ~ rs'
    )
lem-final ss [] ls-halt with safety ss
lem-final ss [] ls-halt | inj₁ (ls-halt' , rs-halt)
  = _ , [] , rs-halt , ss
lem-final ss [] ls-halt | inj₂ (ls' , rs' , (lx :: lxs) , (rx :: rxs) , ss')
  = ⊥-elim (L.final-progressing-absurd ls-halt lx)
lem-final ss (lx :: lxs) ls'-halt with safety ss
lem-final ss (lx :: lxs) ls'-halt | inj₁ (ls-halt , rs-halt)
  = ⊥-elim (L.final-progressing-absurd ls-halt lx)
lem-final ss (lx :: lxs) ls'-halt
```

```
      | inj₂ (ls' , rs' , (lx' ∷ lxs') , (rx' ∷ rxs') , ss')
        with L.deterministic lx lx'
    ... | refl
        with lem-final-helper ss' lxs' lxs (prefix lxs ls'-halt lxs') ls'-halt
    ... | rs'' , rxs , rhalt , bis = rs'' , (rx' ∷ (rxs' R.++ rxs)) , rhalt , bis



module Theorems
  {lState rState : Set}
  (lSystem : System lState)
  (rSystem : System rState)
  (_~_ : lState → rState → Set)
  (safety : {ls : lState}{rs : rState}
    → ls ~ rs
    → (System.Final lSystem ls × System.Final rSystem rs)
      ⊎
      ∃[ ls' ]
      ∃[ rs' ](
        System._⟶+_ lSystem ls ls' ×
        System._⟶+_ rSystem rs rs' ×
        (ls' ~ rs')))
  where

  module L where
    open System lSystem public

  module R where
    open System rSystem public
```

```
thm-final-LR : ∀ {ls rs}
  → (ss : ls ~ rs)
  → ∀ {ls'}
  → ls L.⟶* ls'
  → L.Final ls'
  → ∃[ rs' ](
       rs R.⟶* rs' ×
       R.Final rs' ×
       ls' ~ rs'
    )
thm-final-LR = Lemmas.lem-final lSystem rSystem _~_ safety


safety' : {rs : rState}{ls : lState}
  → ls ~ rs
  → (System.Final rSystem rs × System.Final lSystem ls)
    ⊎
    ∃[ rs' ]
    ∃[ ls' ](
      System._⟶+_ rSystem rs rs' ×
      System._⟶+_ lSystem ls ls' ×
      (ls' ~ rs'))
safety' ss with safety ss
safety' ss | inj₁ (lh , rh) = inj₁ (rh , lh)
safety' ss | inj₂ (ls' , rs' , lxs , rxs , ss')
  = inj₂ (rs' , ls' , rxs , lxs , ss')


thm-final-RL : ∀ {ls rs}
```

```
    → (ss : ls ~ rs)
    → ∀ {rs'}
    → rs R.⟶* rs'
    → R.Final rs'
    → ∃[ ls' ](
      ls L.⟶* ls' ×
      L.Final ls' ×
      ls' ~ rs'
    )
  thm-final-RL
    = Lemmas.lem-final rSystem lSystem (λ r l → l ~ r) safety'


open Theorems public
```

Appendix D

Cast Representations

D.1 Lazy UD Coercions in Normal Form

```
module CastRepresentations.LazyUDCoercionsInNormalForm (Label : Set) where


open import Types renaming (B to tB; _⇒_ to _t⇒_; _⊗_ to _t⊗_)

open import Cast Label using (_⟹[_]_) renaming (Cast to SrcCast)

open import Terms Label

open import S.CastADT Label


open import Relation.Nullary using (Dec; yes; no; ¬_)

open import Data.Empty using (⊥-elim)

open import Relation.Binary.PropositionalEquality using (_≡_; refl; sym; cong)


infix  99 ‘_

infix 100 _⇒_

infix 100 _⊗_


mutual


  data CoeG : PreType → PreType → Set where


    B : CoeG tB tB


    _⇒_ : ∀ {S1 T1 S2 T2}

      → (s : CoeS S2 S1)
```

```
    → (t : CoeS T1 T2)
    → CoeG (S1 t⇒ T1) (S2 t⇒ T2)

  _⊗_ : ∀ {S1 T1 S2 T2}
    → (s : CoeS S1 S2)
    → (t : CoeS T1 T2)
    → CoeG (S1 t⊗ T1) (S2 t⊗ T2)

data CoeI : PreType → Type → Set where

  ⊥ : ∀ {A P Q}
    → (A⌣G : (‘ A) ⌣ (‘ P))
    → (G : Ground P)
    → (l : Label)
    → (H : Ground Q)
    → (¬G≡H : ¬ (P ≡ Q))
    → ∀ {T}
    → CoeI A T

  _,_!! : ∀ {P Q}
    → (g : CoeG P Q)
    → (G : Ground Q)
    → CoeI P ∗

  ‘_ : ∀ {P Q}
    → (g : CoeG P Q)
    → CoeI P (‘ Q)
```

```
  data CoeS : Type → Type → Set where

    id*   : CoeS * *

    _??_,_ : ∀ {P T}
      → (G : Ground P)
      → (l : Label)
      → (i : CoeI P T)
      → CoeS * T

    ‘_    : ∀ {P T}
      → (i : CoeI P T)
      → CoeS (‘ P) T


Cast : Type → Type → Set
Cast = CoeS


lem-g⌣ : ∀ {P Q} → CoeG P Q → (‘ P) ⌣ (‘ Q)
lem-g⌣ B = ⌣B
lem-g⌣ (s ⇒ t) = ⌣⇒
lem-g⌣ (s ⊗ t) = ⌣⊗


mutual
  _g⨟g_ : ∀ {T1 T2 T3} → CoeG T1 T2 → CoeG T2 T3 → CoeG T1 T3
  B g⨟g B = B
  (s1 ⇒ t1) g⨟g (s2 ⇒ t2) = (s2 ⨟ s1) ⇒ (t1 ⨟ t2)
  (s1 ⊗ t1) g⨟g (s2 ⊗ t2) = (s1 ⨟ s2) ⊗ (t1 ⨟ t2)
```

```
  _g⨟i_ : ∀ {T1 T2 T3} → CoeG T1 T2 → CoeI T2 T3 → CoeI T1 T3
  g g⨟i (  h , G !!) =   (g g⨟g h) , G !!
  g g⨟i (‘ h      ) = ‘ (g g⨟g h)
  g g⨟i ⊥ A⌣P G l H ¬P≡Q = ⊥ (⌣trans (lem-g⌣ g) A⌣P) G l H ¬P≡Q

  _i⨟s_ : ∀ {T1 T2 T3} → CoeI T1 T2 → CoeS T2 T3 → CoeI T1 T3
  ⊥ A⌣G G l H ¬G≡H i⨟s s = ⊥ A⌣G G l H ¬G≡H
  (g , G !!) i⨟s id* = (g , G !!)
  (g , G !!) i⨟s (H ?? l , i) with G =?G H
  (g , G !!) i⨟s (H ?? l , i) | yes refl = g g⨟i i
  (g , G !!) i⨟s (H ?? l , i) | no ¬G≡H  = ⊥ (lem-g⌣ g) G l H ¬G≡H
  (‘ g) i⨟s (‘ i) = (g g⨟i i)

  _⨟_ : ∀ {T1 T2 T3} → CoeS T1 T2 → CoeS T2 T3 → CoeS T1 T3
  id* ⨟ t = t
  (G ?? l , i) ⨟ t = G ?? l , (i i⨟s t)
  (‘       i) ⨟ t = ‘       (i i⨟s t)

mutual
  ⇑* : Label → ∀ T → Cast T *
  ⇑* l *     = id*
  ⇑* l (‘ P) = ⇑ l P

  ⇑ : Label → ∀ P → Cast (‘ P) *
  ⇑ l tB       = ‘ (B , ‘B !!)
  ⇑ l (S t⇒ T) = ‘ (⇓* l S ⇒ ⇑* l T , ‘⇒ !!)
  ⇑ l (S t⊗ T) = ‘ (⇑* l S ⊗ ⇑* l T , ‘⊗ !!)
```

```
  ⇓* : Label → ∀ T → Cast ∗ T
  ⇓* l ∗     = id∗
  ⇓* l (‘ P) = ⇓ l P

  ⇓ : Label → ∀ P → Cast ∗ (‘ P)
  ⇓ l tB       = (‘B ?? l , ‘ B)
  ⇓ l (S t⇒ T) = (‘⇒ ?? l , ‘ ⇑* l S ⇒ (⇓* l T))
  ⇓ l (S t⊗ T) = (‘⊗ ?? l , ‘ ⇓* l S ⊗ (⇓* l T))

lem-¬⌣-ground : {P Q : PreType}
  → ¬ (‘ P) ⌣ (‘ Q)
  → ¬ (ground P ≡ ground Q)
lem-¬⌣-ground {tB} {tB} ¬p = λ _ → ¬p ⌣B
lem-¬⌣-ground {tB} {S t⇒ T} ¬p = λ ()
lem-¬⌣-ground {tB} {S t⊗ T} ¬p = λ ()
lem-¬⌣-ground {S t⇒ T} {tB} ¬p = λ ()
lem-¬⌣-ground {S t⇒ T} {S₁ t⇒ T₁} ¬p = λ _ → ¬p ⌣⇒
lem-¬⌣-ground {S t⇒ T} {S₁ t⊗ T₁} ¬p = λ ()
lem-¬⌣-ground {S t⊗ T} {tB} ¬p = λ ()
lem-¬⌣-ground {S t⊗ T} {S₁ t⇒ T₁} ¬p = λ ()
lem-¬⌣-ground {S t⊗ T} {S₁ t⊗ T₁} ¬p = λ _ → ¬p ⌣⊗

⌈_⌉ : ∀ {T1 T2} → SrcCast T1 T2 → Cast T1 T2
⌈ *   ⟹[ l ] *   ⌉ = id∗
⌈ *   ⟹[ l ] ‘ Q ⌉ = ⇓ l Q
⌈ ‘ P ⟹[ l ] *   ⌉ = ⇑ l P
⌈ ‘ P ⟹[ l ] ‘ Q ⌉ with (‘ P) ⌣? (‘ Q)
⌈ ‘ P ⟹[ l ] ‘ Q ⌉             | no ¬P⌣Q
```

```
  = ‘ (⊥ (ground-⌣ P) (ground-Ground P) l (ground-Ground Q) (lem-¬⌣-ground ¬P⌣Q))
⌈ ‘ tB       ⟹[ l ] ‘ tB       ⌉ | yes ⌣B = ‘ (‘ B)
⌈ ‘ S1 t⇒ T1 ⟹[ l ] ‘ S2 t⇒ T2 ⌉ | yes ⌣⇒
  = ‘ (‘ ⌈ S2 ⟹[ l ] S1 ⌉ ⇒ ⌈ T1 ⟹[ l ] T2 ⌉)
⌈ ‘ L1 t⊗ R1 ⟹[ l ] ‘ L2 t⊗ R2 ⌉ | yes ⌣⊗
  = ‘ (‘ ⌈ L1 ⟹[ l ] L2 ⌉ ⊗ ⌈ R1 ⟹[ l ] R2 ⌉)



mutual
  id : ∀ T → CoeS T T
  id ∗
    = id∗
  id (‘ P)
    = ‘ (‘ id-g P)


  id-g : ∀ P → CoeG P P
  id-g tB
    = B
  id-g (S t⇒ T)
    = id S ⇒ id T
  id-g (S t⊗ T)
    = (id S) ⊗ (id T)


open import R.BlameStrategies Label using (BlameStrategy; LazyUDBS)
open BlameStrategy LazyUDBS using (Injectable)


open import S.Values Label Injectable Cast
```

```
open import Error
  using (Error; return; raise; _>>=_; _>=>_
        ;>>=-return; >>=-assoc; >=>-assoc; >=>->>=)


CastResult : Type → Set
CastResult T = Error Label (Value T)


⟦_⟧g : ∀ {P Q}
  → CoeG P Q
  → Value (‘ P)
  → Value (‘ Q)
⟦ B ⟧g v = v
⟦ s2 ⇒ t2 ⟧g (lam⟨ s1 ⇒ t1 ⟩ e E) = lam⟨ s2 ⨟ s1 ⇒ t1 ⨟ t2 ⟩ e E
⟦ s2 ⊗ t2 ⟧g (cons⟨ s1 ⊗ t1 ⟩ v u) = cons⟨ s1 ⨟ s2 ⊗ t1 ⨟ t2 ⟩ v u


⟦_⟧i : ∀ {P T}
  → CoeI P T
  → Value (‘ P)
  → CastResult T
⟦ g , G !! ⟧i v = return ((dyn G) (⟦ g ⟧g v))
⟦ ‘ g     ⟧i v = return (⟦ g ⟧g v)
⟦ ⊥ A⌣G G l H ¬G≡H ⟧i v = raise l


project : ∀ {P}
  → Ground P
  → Label
  → Value ⋆
  → CastResult (‘ P)
```

```
project H l (dyn G v) with G =?G H

project H l (dyn G v) | yes refl = return v

project H l (dyn G v) | no ¬G≡H  = raise l


⟦_⟧ : ∀ {S T}

  → Cast S T

  → Value S

  → CastResult T

⟦ id*       ⟧ v = return v

⟦ G ?? l , i ⟧ v = project G l v >>= ⟦ i ⟧i

⟦ ‘ i       ⟧ v = ⟦ i ⟧i v


mutual

  g-identity^l : ∀ {P1 P2} → (g : CoeG P1 P2) → id-g P1 g⨟g g ≡ g

  g-identity^l B = refl

  g-identity^l (s ⇒ t) rewrite identity^r s | identity^l t = refl

  g-identity^l (s ⊗ t) rewrite identity^l s | identity^l t = refl


  g-identity^r : ∀ {P1 P2} → (g : CoeG P1 P2) → g g⨟g id-g P2 ≡ g

  g-identity^r B = refl

  g-identity^r (s ⇒ t) rewrite identity^l s | identity^r t = refl

  g-identity^r (s ⊗ t) rewrite identity^r s | identity^r t = refl


  i-identity^r : ∀ {P1 P2} → (i : CoeI P1 P2) → i i⨟s id P2 ≡ i

  i-identity^r (g , G !!) = refl

  i-identity^r (‘ g) rewrite g-identity^r g = refl

  i-identity^r (⊥ A⌣G G l H ¬G≡H) = refl
```

```
  identity^l : ∀ {T1 T2} → (c : Cast T1 T2) → id T1 ⨟ c ≡ c
  identity^l id* = refl
  identity^l (G ?? l , i) = refl
  identity^l (‘ (g , G !!)) rewrite g-identity^l g = refl
  identity^l (‘ (‘ g))     rewrite g-identity^l g = refl
  identity^l (‘ ⊥ A⌣G G l H ¬G≡H) = cong (λ □ → (‘ ⊥ □ G l H ¬G≡H)) (⌣unique _ _)

  identity^r : ∀ {T1 T2} → (c : Cast T1 T2) → c ⨟ id T2 ≡ c
  identity^r id* = refl
  identity^r (G ?? l , i) rewrite i-identity^r i = refl
  identity^r (‘ i) rewrite i-identity^r i = refl

mutual
  assoc-ggg : ∀ {T1 T2 T3 T4}
    → (c1 : CoeG T1 T2)
    → (c2 : CoeG T2 T3)
    → (c3 : CoeG T3 T4)
    → (c1 g⨟g c2) g⨟g c3 ≡ c1 g⨟g (c2 g⨟g c3)
  assoc-ggg B B B = refl
  assoc-ggg (s1 ⇒ t1) (s2 ⇒ t2) (s3 ⇒ t3)
    rewrite assoc s3 s2 s1 | assoc t1 t2 t3
    = refl
  assoc-ggg (s1 ⊗ t1) (s2 ⊗ t2) (s3 ⊗ t3)
    rewrite assoc s1 s2 s3 | assoc t1 t2 t3
    = refl

  assoc-ggi : ∀ {T1 T2 T3 T4}
    → (c1 : CoeG T1 T2)
```

```
  → (c2 : CoeG T2 T3)
  → (c3 : CoeI T3 T4)
  → (c1 g⨟g c2) g⨟i c3 ≡ c1 g⨟i (c2 g⨟i c3)
assoc-ggi g1 g2 (⊥ A⌣G G l H ¬G≡H) = cong (λ □ → ⊥ □ G l H ¬G≡H) (⌣unique _ _)
assoc-ggi g1 g2 (g , G !!)
  rewrite assoc-ggg g1 g2 g = refl
assoc-ggi g1 g2 (` g)
  rewrite assoc-ggg g1 g2 g = refl

assoc-gis : ∀ {T1 T2 T3 T4}
  → (c1 : CoeG T1 T2)
  → (c2 : CoeI T2 T3)
  → (c3 : CoeS T3 T4)
  → (c1 g⨟i c2) i⨟s c3 ≡ c1 g⨟i (c2 i⨟s c3)
assoc-gis g1 (⊥ A⌣G G l H ¬G≡H) s = refl
assoc-gis g1 (g2 , G !!) id* = refl
assoc-gis g1 (g2 , G !!) (H ?? l , i3) with G =?G H
assoc-gis g1 (g2 , G !!) (H ?? l , i3) | no ¬p
  = cong (λ □ → ⊥ □ G l H ¬p) (⌣unique _ _)
assoc-gis g1 (g2 , G !!) (H ?? l , i3) | yes refl
  rewrite assoc-ggi g1 g2 i3 = refl
assoc-gis g1 (` g2) (` i3)
  rewrite assoc-ggi g1 g2 i3 = refl

assoc-iss : ∀ {T1 T2 T3 T4}
  → (c1 : CoeI T1 T2)
  → (c2 : Cast T2 T3)
  → (c3 : Cast T3 T4)
```

```
    → (c1 i⨟s c2) i⨟s c3 ≡ c1 i⨟s (c2 ⨟ c3)
  assoc-iss (⊥ A⌣G G l H ¬G≡H) s1 s2 = refl
  assoc-iss (g , G !!) id* s2 = refl
  assoc-iss (g , G !!) (H ?? l , i) s2 with G =?G H
  assoc-iss (g , G !!) (H ?? l , i) s2 | no ¬G≡H  = refl
  assoc-iss (g , G !!) (H ?? l , i) s2 | yes refl rewrite assoc-gis g i s2 = refl
  assoc-iss (‘ g) (‘ i) s2 rewrite assoc-gis g i s2 = refl


  assoc : ∀ {T1 T2 T3 T4}
    → (c1 : Cast T1 T2)
    → (c2 : Cast T2 T3)
    → (c3 : Cast T3 T4)
    → (c1 ⨟ c2) ⨟ c3 ≡ c1 ⨟ (c2 ⨟ c3)
  assoc id* c2 c3 = refl
  assoc (G ?? l , i) c2 c3 rewrite assoc-iss i c2 c3 = refl
  assoc (‘ i) c2 c3 rewrite assoc-iss i c2 c3 = refl


lem-id : ∀ {T}
  → (v : Value T)
  ----------------------------
  → ⟦ id T ⟧ v ≡ return v
lem-id {*} v = refl
lem-id {‘ tB} v = refl
lem-id {‘ S t⇒ T} (lam⟨ s ⇒ t ⟩ e E)
  rewrite identity^l s | identity^r t
  = refl
lem-id {‘ S t⊗ T} (cons⟨ s ⊗ t ⟩ v u)
  rewrite identity^r s | identity^r t
```

```
  = refl


lem-g⨟g : ∀ {T1 T2 T3}
  → (c1 : CoeG T1 T2)
  → (c2 : CoeG T2 T3)
  → ∀ v
  --------------------
  → ⟦ c1 g⨟g c2 ⟧g v ≡ ⟦ c2 ⟧g (⟦ c1 ⟧g v)
lem-g⨟g B B v = refl
lem-g⨟g (s2 ⇒ t2) (s3 ⇒ t3) (lam⟨ s1 ⇒ t1 ⟩ e E)
  rewrite assoc s3 s2 s1 | assoc t1 t2 t3
  = refl
lem-g⨟g (s2 ⊗ t2) (s3 ⊗ t3) (cons⟨ s1 ⊗ t1 ⟩ v u)
  rewrite assoc s1 s2 s3 | assoc t1 t2 t3
  = refl


lem-g⨟i : ∀ {T1 T2 T3}
  → (c1 : CoeG T1 T2)
  → (c2 : CoeI T2 T3)
  → ∀ v
  --------------------
  → ⟦ c1 g⨟i c2 ⟧i v ≡ (return (⟦ c1 ⟧g v)) >>= ⟦ c2 ⟧i
lem-g⨟i g1 (g2 , G !!) v rewrite lem-g⨟g g1 g2 v = refl
lem-g⨟i g1 (‘ g2)     v rewrite lem-g⨟g g1 g2 v = refl
lem-g⨟i g1 (⊥ A⌣G G l H ¬G≡H) v = refl


lem-i⨟s : ∀ {T1 T2 T3}
  → (c1 : CoeI T1 T2)
```

```
  → (c2 : CoeS T2 T3)
  → ∀ v
  --------------------
  → ⟦ c1 ⨟s c2 ⟧i v ≡ ⟦ c1 ⟧i v >>= ⟦ c2 ⟧
lem-⨟s (g , G !!) id* v = refl
lem-⨟s (g , G !!) (H ?? l , i) v with G =?G H
lem-⨟s (g , G !!) (H ?? l , i) v | yes refl = lem-g⨟i g i v
lem-⨟s (g , G !!) (H ?? l , i) v | no ¬G≡H  = refl
lem-⨟s (` g) (` i) v = lem-g⨟i g i v
lem-⨟s (⊥ A⌣G G l H ¬G≡H) s v = refl


lem-seq : ∀ {T1 T2 T3}
  → (c1 : Cast T1 T2)
  → (c2 : Cast T2 T3)
  → ∀ v
  --------------------
  → ⟦ c1 ⨟ c2 ⟧ v ≡ ⟦ c1 ⟧ v >>= ⟦ c2 ⟧
lem-seq id* t v = refl
lem-seq (G1 ?? l1 , i1) t v with project G1 l1 v
lem-seq (G1 ?? l1 , i1) t v | raise  l' = refl
lem-seq (G1 ?? l1 , i1) t v | return v' = lem-⨟s i1 t v'
lem-seq (` i1) t v = lem-⨟s i1 t v


S : CastADT Injectable
S = record
    { Cast = Cast
    ; id  = id
    ; ⌈_⌉ = ⌈_⌉
```

```
    ; _⨟°_ = _⨟°_
    ; ⟦_⟧ = ⟦_⟧
    ; lem-id = λ T v → lem-id v
    ; lem-seq = lem-seq
    }

open import S.LazyUDCastADT Label

eq-¬⌣ : ∀ {T1 T2}
  → (v : Value T1)
  → (l : Label)
  → ¬ (T1 ⌣ T2)
  ---
  → ⟦ ⌈ T1 ⟹[ l ] T2 ⌉ ⟧ v
      ≡
    raise l
eq-¬⌣ {*} {*} v l ¬p = ⊥-elim (¬p *⌣*)
eq-¬⌣ {*} {‘ P} v l ¬p = ⊥-elim (¬p (*⌣P P))
eq-¬⌣ {‘ P} {*} v l ¬p = ⊥-elim (¬p (P⌣* P))
eq-¬⌣ {‘ P} {‘ Q} v l ¬p with (‘ P) ⌣? (‘ Q)
eq-¬⌣ {‘ P} {‘ Q} v l ¬p | yes p’ = ⊥-elim (¬p p’)
eq-¬⌣ {‘ P} {‘ Q} v l ¬p | no ¬p’ = refl

lem-⇑* : (l : Label)(T : Type)
  → (⇑* l T) ≡ ⌈ T ⟹[ l ] * ⌉
lem-⇑* l * = refl
lem-⇑* l (‘ P) = refl
```

```
lem-⇓* : (l : Label)(T : Type)
  → (⇓* l T) ≡ ⌈ * ⟹[ l ] T ⌉
lem-⇓* l * = refl
lem-⇓* l (‘ P) = refl


lem-⇑ : (l : Label)(P : PreType)
  → (⇑ l P) ≡ (⌈ (‘ P) ⟹[ l ] ‘ ground P ⌉ ⨟ ⌈ ‘ ground P ⟹[ l ] * ⌉)
lem-⇑ l tB = refl
lem-⇑ l (S t⇒ T)
  rewrite lem-⇓* l S | lem-⇑* l T
    | identityʳ ⌈ T ⟹[ l ] * ⌉
  = refl
lem-⇑ l (S t⊗ T)
  rewrite lem-⇑* l S | lem-⇑* l T
    | identityʳ ⌈ T ⟹[ l ] * ⌉
    | identityʳ ⌈ S ⟹[ l ] * ⌉
  = refl


lem-⇓ : (l : Label)(P : PreType)
  → (⇓ l P) ≡ (⌈ * ⟹[ l ] ‘ ground P ⌉ ⨟ ⌈ ‘ ground P ⟹[ l ] ‘ P ⌉)
lem-⇓ l tB = refl
lem-⇓ l (S t⇒ T)
  rewrite lem-⇑* l S | lem-⇓* l T
    | identityʳ ⌈ S ⟹[ l ] * ⌉
  = refl
lem-⇓ l (S t⊗ T)
  rewrite lem-⇓* l S | lem-⇓* l T
  = refl
```

```
eq-P⋆ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → ¬ Ground P
  → ⟦ ⌈ (‘ P) ⟹[ l ] ⋆ ⌉ ⟧ v
      ≡
    ⟦ ⌈ (‘ P) ⟹[ l ] (‘ ground P) ⌉ ⟧ v >>= ⟦ ⌈ (‘ ground P) ⟹[ l ] ⋆ ⌉ ⟧
eq-P⋆ {P} v l ¬gP
  rewrite lem-⇑ l P
  | lem-seq ⌈ (‘ P) ⟹[ l ] (‘ ground P) ⌉ ⌈ (‘ ground P) ⟹[ l ] ⋆ ⌉ v
  = refl


eq-I⋆ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → ⟦ ⌈ ‘ P ⟹[ l ] ⋆ ⌉ ⟧ v
      ≡
    return (dyn gP v)
eq-I⋆ {.tB} v l ‘B = refl
eq-I⋆ {.(⋆ t⇒ ⋆)} (lam⟨ c1 ⇒ c2 ⟩ e E) l ‘⇒
  rewrite identity^r c2
  = refl
eq-I⋆ {.(⋆ t⊗ ⋆)} (cons⟨ c1 ⊗ c2 ⟩ v v₁) l ‘⊗
  rewrite identity^r c1 | identity^r c2
  = refl
```

```
eq-⋆P : ∀ {P}
  → (v : Value ⋆)
  → (l : Label)
  → ¬ Ground P
  → ⟦ ⌈ ⋆ ⟹[ l ] (‘ P) ⌉ ⟧ v
      ≡
    ⟦ ⌈ ⋆ ⟹[ l ] (‘ ground P) ⌉ ⟧ v >>= ⟦ ⌈ (‘ ground P) ⟹[ l ] (‘ P) ⌉ ⟧
eq-⋆P {P} v l ¬gP
  rewrite lem-⇓ l P
  | lem-seq ⌈ ⋆ ⟹[ l ] (‘ ground P) ⌉ ⌈ (‘ ground P) ⟹[ l ] (‘ P) ⌉ v
  = refl


eq-⋆I-succ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → ⟦ ⌈ ⋆ ⟹[ l ] (‘ P) ⌉ ⟧ (dyn gP v)
      ≡
    return v
eq-⋆I-succ v l ‘B = refl
eq-⋆I-succ (lam⟨ c1 ⇒ c2 ⟩ e E) l ‘⇒
  rewrite identityʳ c2
  = refl
eq-⋆I-succ (cons⟨ c1 ⊗ c2 ⟩ v v₁) l ‘⊗
  rewrite identityʳ c1 | identityʳ c2
  = refl


eq-⋆I-fail : {P Q : PreType}
```

```
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → (gQ : Ground Q)
  → ¬ (_≡_ {A = Type} (‘ P) (‘ Q))
  → ⟦ ⌈ * ⟹[ l ] (‘ Q) ⌉ ⟧ (dyn gP v)
      ≡
    raise l
eq-*I-fail {tB} v l ‘B ‘B ¬p = ⊥-elim (¬p refl)
eq-*I-fail {tB} v l ‘B ‘⇒ ¬p = refl
eq-*I-fail {tB} v l ‘B ‘⊗ ¬p = refl
eq-*I-fail {.* t⇒ .*} v l ‘⇒ ‘B ¬p = refl
eq-*I-fail {.* t⇒ .*} v l ‘⇒ ‘⇒ ¬p = ⊥-elim (¬p refl)
eq-*I-fail {.* t⇒ .*} v l ‘⇒ ‘⊗ ¬p = refl
eq-*I-fail {.* t⊗ .*} v l ‘⊗ ‘B ¬p = refl
eq-*I-fail {.* t⊗ .*} v l ‘⊗ ‘⇒ ¬p = refl
eq-*I-fail {.* t⊗ .*} v l ‘⊗ ‘⊗ ¬p = ⊥-elim (¬p refl)


S-LazyUD : LazyUD S
S-LazyUD = record
             { eq-¬⌣ = eq-¬⌣
             ; eq-** = λ l v → refl
             ; eq-P* = eq-P*
             ; eq-I* = eq-I*
             ; eq-*P = eq-*P
             ; eq-*I-succ = eq-*I-succ
             ; eq-*I-fail = eq-*I-fail
             ; eq-B = λ l b → refl
```

```
            ; eq-⇒ = λ T21 T22 T11 T12 {S} {T} l {Γ} c₁ c₂ e E → refl
            ; eq-⊗ = λ T21 T22 T11 T12 {S} {T} l c₁ c₂ v1 v2 → refl
            }
```

D.2 Lazy UD Threesomes

```
module CastRepresentations.LazyUDThreesomes (Label : Set) where


open import Types
open import Cast Label using (_⟹[_]_) renaming (Cast to SrcCast)
open import Terms Label
open import S.CastADT Label


open import Data.Empty using () renaming (⊥ to Empty)
open import Data.Empty.Irrelevant using (⊥-elim)
open import Data.Product using (Îč-syntax; ∃-syntax; proj₁; proj₂; _×_; _,_)
open import Data.Vec using (Vec; []; _∷_; replicate)
open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Relation.Binary.PropositionalEquality using (_≡_; refl; sym; cong)
open Relation.Binary.PropositionalEquality.≡-Reasoning
  using (begin_; _≡⟨⟩_; _≡⟨_⟩_; _ □)


{-
Siek, Jeremy G., and Philip Wadler. "Threesomes, with and without blame."
ACM Sigplan Notices 45.1 (2010): 365-376.
-}


{-
Ground Types are isomorphic to Type Operators in the obvious way. To make
```

```
proof easier, we shall use the two sets interchangablely in this module.
-}

data OptionalLabel (op : TypeOp) : Type → (Vec Type (arity op)) → Set where
  ?? : (l : Label) → OptionalLabel op ∗ (replicate ∗)
  ϵ : ∀ {Ts} → OptionalLabel op (‘ op · Ts) Ts

data Tail (op : TypeOp) : (Vec Type (arity op)) → Type → Set where
  !! : Tail op (replicate ∗) ∗
  ϵ : ∀ {Ts} → Tail op Ts (‘ op · Ts)

data LabeledType : Type → Type → Set
data LabeledPreType : (op : TypeOp) → (Ss Ts : Vec Type (arity op)) → Set

data LabeledType where

  ∗ : LabeledType ∗ ∗

  ⊥ : ∀ {T S}
    → (l : Label)
    → (G : TypeOp)
    → ∀ {Ss}
    → (p : OptionalLabel G S Ss)
    → LabeledType S T

  ^ : ∀ {S G Ss Ts T}
    → (P̂ : LabeledPreType G Ss Ts)
    → (p : OptionalLabel G S Ss)
```

```
    → {t : Tail G Ts T}
      ---------------
    → LabeledType S T

data LabeledPreType where

  B̂ : LabeledPreType ‘B [] []

  _⇛̂_ : ∀ {S1 S2 T1 T2}
    → (Ŝ : LabeledType S2 S1)
    → (T̂ : LabeledType T1 T2)
    → LabeledPreType ‘⇒ (S1 :: T1 :: []) (S2 :: T2 :: [])

  _⊗̂_ : ∀ {S1 S2 T1 T2}
    → (Ŝ : LabeledType S1 S2)
    → (T̂ : LabeledType T1 T2)
    → LabeledPreType ‘⊗ (S1 :: T1 :: []) (S2 :: T2 :: [])

-- In real implementation, one should use the gnd function
-- But here we use quick-gnd to simplify the proof.
-- Functions gnd and quick-gnd are equivalent, as shown in gnd-faithful

quick-gnd : ∀ {op Ss Ts} → (P̂ : LabeledPreType op Ss Ts) → TypeOp
quick-gnd {op} P̂ = op

gnd : ∀ {op Ss Ts} → (P̂ : LabeledPreType op Ss Ts) → TypeOp
gnd B̂ = ‘B
gnd (Ŝ ⇛̂ T̂) = ‘⇒
```

```
gnd (Ŝ ⊗̂ T̂) = ‘⊗


gnd-faithful : ∀ {op Ss Ts} → (P̂ : LabeledPreType op Ss Ts) → gnd P̂ ≡ quick-gnd P̂

gnd-faithful B̂ = refl

gnd-faithful (Ŝ ⇒̂ T̂) = refl

gnd-faithful (Ŝ ⊗̂ T̂) = refl


impossible-tail : ∀ {G H Ss Ts}

  → Tail G  Ss (‘ H · Ts)

  → ¬ G ≡ H

  → Empty

impossible-tail ϵ ¬G≡H = ¬G≡H refl


no-gap-at-all : ∀ {G Ss T Ts}

  → Tail G Ss T

  → OptionalLabel G T Ts

  → Ss ≡ Ts

no-gap-at-all !! (?? l) = refl

no-gap-at-all ϵ ϵ = refl


_○_ : ∀ {T1 T2 T3} → LabeledType T2 T3 → LabeledType T1 T2 → LabeledType T1 T3

T̂ ○ ∗          = T̂

T̂ ○ ⊥ m G p    = ⊥ m G p

∗ ○ Ŝ@(^ P̂ p) = Ŝ

⊥ m H q      ○ ^ P̂ p with quick-gnd P̂ =?op H

⊥ m H q      ○ ^ P̂ p | yes refl = ⊥ m H p

⊥ m H (?? l) ○ ^ P̂ p | no ¬G≡H  = ⊥ l (quick-gnd P̂) p

⊥ m H ϵ      ○ ^ P̂ p {t} | no ¬G≡H  = ⊥-elim (impossible-tail t ¬G≡H)
```

```
^ Q̂ q     ∘ ^ P̂ p {t} with quick-gnd P̂ =?op quick-gnd Q̂
^ Q̂ (?? m) ∘ ^ P̂ p {t} | no ¬G≡H = ⊥ m (quick-gnd P̂) p
^ Q̂ ϵ     ∘ ^ P̂ p {t} | no ¬G≡H = ⊥-elim (impossible-tail t ¬G≡H)
^ Q̂ q     ∘ ^ P̂ p {t} | yes refl with no-gap-at-all t q
^ B̂         q {s} ∘ ^ B̂         p {t} | yes refl | refl = ^ B̂ p {s}
^ (Ŝ₂ ⇛ T̂₂) q {s} ∘ ^ (Ŝ₁ ⇛ T̂₁) p {t} | yes refl | refl
  = ^ ((Ŝ₁ ∘ Ŝ₂) ⇛ (T̂₂ ∘ T̂₁)) p {s}
^ (Ŝ₂ ⊗̂ T̂₂) q {s} ∘ ^ (Ŝ₁ ⊗̂ T̂₁) p {t} | yes refl | refl
  = ^ ((Ŝ₂ ∘ Ŝ₁) ⊗̂ (T̂₂ ∘ T̂₁)) p {s}


fail-left : ∀ {T1 T2}
  → (l : Label)
  → (G : TypeOp)
  → ∀ {Ss}
  → (p : OptionalLabel G T2 Ss)
  → (T̂ : LabeledType T1 T2)
  → ∃[ l' ]
    (∃[ G' ]
    (∃[ Ss ]
    (Îč[ p' âĹĹ OptionalLabel G' T1 Ss ]
    (∀ {T3} → (⊥ {T3} l G p ∘ T̂) ≡ ⊥ {T3} l' G' p' ))))
fail-left l G p ∗ = _ , _ , _ , _ , refl
fail-left l G p (⊥ l₁ G₁ p₁) = _ , _ , _ , _ , refl
fail-left l G p (^ P̂ p₁) with quick-gnd P̂ =?op G
fail-left l G p (^ P̂ p₁) | yes refl = _ , _ , _ , _ , refl
fail-left l G (?? l₁) (^ P̂ p₁) | no ¬p = _ , _ , _ , _ , refl
fail-left l G ϵ (^ P̂ p₁ {t}) | no ¬p = ⊥-elim (impossible-tail t ¬p)
```

```
dyn-left : ∀ {T}
  → (T̂ : LabeledType T ⋆)
  → ⋆ ∘ T̂ ≡ T̂
dyn-left ⋆ = refl
dyn-left (⊥ l G p) = refl
dyn-left (^ P̂ p) = refl


dec-yes : {A : Set}{a : A}(a=?a : Dec (a ≡ a)) → ∃[ p ](a=?a ≡ yes p)
dec-yes (yes p) = p , refl
dec-yes (no ¬p) = ⊥-elim (¬p refl)


∘-assoc : ∀ {T1 T2 T3 T4}
  → (T̂₃ : LabeledType T3 T4)
  → (T̂₂ : LabeledType T2 T3)
  → (T̂₁ : LabeledType T1 T2)
  → (T̂₃ ∘ (T̂₂ ∘ T̂₁)) ≡ ((T̂₃ ∘ T̂₂) ∘ T̂₁)
∘-assoc T̂₃ T̂₂ ⋆ = refl
∘-assoc T̂₃ T̂₂ (⊥ l₁ G₁ p₁) = refl
∘-assoc T̂₃ ⋆ (^ P̂₁ p₁) = refl
∘-assoc {T3 = T3}{T4 = T4} T̂₃ T̂₂@(⊥ l₂ G₂ p₂) T̂₁@(^ P̂₁ p₁)
  with fail-left l₂ G₂ p₂ T̂₁
... | _ , _ , _ , _ , eq
  rewrite eq {T3} | eq {T4} = refl
∘-assoc ⋆ T̂₂@(^ P̂₂ p₂) T̂₁@(^ P̂₁ p₁) rewrite dyn-left (T̂₂ ∘ T̂₁) = refl
∘-assoc (⊥ l₃ G₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  with quick-gnd P̂₁ =?op quick-gnd P̂₂
∘-assoc (⊥ l₃ G₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁}) | yes G₁≡G₂
  with quick-gnd P̂₂ =?op G₃
```

```
∘-assoc (⊥ l₃ G₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁}) | yes refl | yes refl
  with no-gap-at-all t₁ p₂
∘-assoc (⊥ l₃ ._‘B p₃) (^ B̂ p₂ {t₂}) (^ B̂ p₁ {t₁})
  | yes refl | yes refl | refl = refl
∘-assoc (⊥ l₃ ._‘⇒ p₃) (^ (Ŝ₁ ⇛ T̂₁) p₂ {t₂}) (^ (Ŝ ⇛ T̂) p₁ {t₁})
  | yes refl | yes refl | refl = refl
∘-assoc (⊥ l₃ ._‘⊗ p₃) (^ (Ŝ₁ ⊗̂ T̂₁) p₂ {t₂}) (^ (Ŝ ⊗̂ T̂) p₁ {t₁})
  | yes refl | yes refl | refl = refl
∘-assoc (⊥ l₃ G₃ (?? l)) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | yes G₁≡G₂ | no ¬G₂≡G₃
  with quick-gnd P̂₁ =?op quick-gnd P̂₂
∘-assoc (⊥ l₃ G₃ (?? l)) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl
  with no-gap-at-all t₁ p₂
∘-assoc (⊥ l₃ ‘B (?? l)) (^ B̂ p₂ {t₂}) (^ B̂ p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = ⊥-elim (¬G₂≡G₃ refl)
∘-assoc (⊥ l₃ ‘⊗ (?? l)) (^ B̂ p₂ {t₂}) (^ B̂ p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = refl
∘-assoc (⊥ l₃ ‘⇒ (?? l)) (^ B̂ p₂ {t₂}) (^ B̂ p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = refl
∘-assoc (⊥ l₃ ‘B (?? l)) (^ (Ŝ₂ ⇛ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⇛ T̂₁) p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = refl
∘-assoc (⊥ l₃ ‘⊗ (?? l)) (^ (Ŝ₂ ⇛ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⇛ T̂₁) p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = refl
∘-assoc (⊥ l₃ ‘⇒ (?? l)) (^ (Ŝ₂ ⇛ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⇛ T̂₁) p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = ⊥-elim (¬G₂≡G₃ refl)
∘-assoc (⊥ l₃ ‘B (?? l)) (^ (Ŝ₂ ⊗̂ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⊗̂ T̂₁) p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = refl
```

```
∘-assoc (⊥ l₃ ‘⊗ (?? l)) (^ (Ŝ₂ ⊗̂ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⊗̂ T̂₁) p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = ⊥-elim (¬G₂≡G₃ refl)
∘-assoc (⊥ l₃ ‘⇒ (?? l)) (^ (Ŝ₂ ⊗̂ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⊗̂ T̂₁) p₁ {t₁})
  | yes refl | no ¬G₂≡G₃ | yes refl | refl = refl
∘-assoc (⊥ l₃ G₃ (?? l)) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | yes G₁≡G₂ | no ¬G₂≡G₃ | no ¬G₁≡G₂' = ⊥-elim (¬G₁≡G₂' G₁≡G₂)
∘-assoc (⊥ l₃ G₃ ϵ) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | yes G₁≡G₂ | no ¬G₂≡G₃ = ⊥-elim (impossible-tail t₂ ¬G₂≡G₃)
∘-assoc (⊥ l₃ G₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂
  with quick-gnd P̂₂ =?op G₃
∘-assoc (⊥ l₃ G₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂
  | yes refl
  with quick-gnd P̂₁ =?op quick-gnd P̂₂
∘-assoc (⊥ l₃ G₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl
  | yes G₁≡G₂' = ⊥-elim (¬G₁≡G₂ G₁≡G₂')
∘-assoc (⊥ l₃ G₃ p₃) (^ P̂₂ (?? l) {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | no ¬G₁≡G₂' = refl
∘-assoc (⊥ l₃ G₃ p₃) (^ P̂₂ ϵ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | no ¬G₁≡G₂' = ⊥-elim (impossible-tail t₁ ¬G₁≡G₂)
∘-assoc (⊥ l₃ G₃ (?? l)) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | no ¬G₂≡G₃
  with quick-gnd P̂₁ =?op quick-gnd P̂₂
∘-assoc (⊥ l₃ G₃ (?? l)) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | no ¬G₂≡G₃ | yes G₁≡G₂' = ⊥-elim (¬G₁≡G₂ G₁≡G₂')
∘-assoc (⊥ l₃ G₃ (?? l)) (^ P̂₂ (?? l₁) {t₂}) (^ P̂₁ p₁ {t₁})
```

```
  | no ¬G₁≡G₂ | no ¬G₂≡G₃ | no ¬G₁≡G₂' = refl
∘-assoc (⊥ l₃ G₃ (?? l)) (^ P̂₂ ϵ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | no ¬G₂≡G₃ | no ¬G₁≡G₂' = ⊥-elim (impossible-tail t₁ ¬G₁≡G₂)
∘-assoc (⊥ l₃ G₃ ϵ) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | no ¬G₂≡G₃ = ⊥-elim (impossible-tail t₂ ¬G₂≡G₃)
-- ∘-assoc (^ P̂₃ p₃) (^ P̂₂ p₂) (^ P̂₁ p₁) = {!!}
∘-assoc (^ P̂₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  with quick-gnd P̂₁ =?op quick-gnd P̂₂
∘-assoc (^ P̂₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁}) | yes refl
  with no-gap-at-all t₁ p₂
∘-assoc (^ B̂ p₃) (^ B̂ p₂ {t₂}) (^ B̂ p₁ {t₁}) | yes refl | refl
  with no-gap-at-all t₂ p₃
... | refl
  with no-gap-at-all t₁ p₂
... | refl = refl
∘-assoc (^ (Ŝ ⇛ T̂) (?? l)) (^ B̂ p₂ {t₂}) (^ B̂ p₁ {t₁}) | yes refl | refl = refl
∘-assoc (^ (Ŝ ⊗̂ T̂) (?? l)) (^ B̂ p₂ {t₂}) (^ B̂ p₁ {t₁}) | yes refl | refl = refl
∘-assoc (^ B̂ (?? l)) (^ (Ŝ₂ ⇛ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⇛ T̂₁) p₁ {t₁})
  | yes refl | refl = refl
∘-assoc (^ (Ŝ₃ ⇛ T̂₃) p₃) (^ (Ŝ₂ ⇛ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⇛ T̂₁) p₁ {t₁})
  | yes refl | refl
  with no-gap-at-all t₂ p₃
... | refl
  with no-gap-at-all t₁ p₂
... | refl
  rewrite ∘-assoc Ŝ₁ Ŝ₂ Ŝ₃ | ∘-assoc T̂₃ T̂₂ T̂₁ = refl
∘-assoc (^ (Ŝ ⊗̂ T̂) (?? l)) (^ (Ŝ₂ ⇛ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⇛ T̂₁) p₁ {t₁})
  | yes refl | refl = refl
```

```
∘-assoc (^ B̂ (?? l)) (^ (Ŝ₂ ⊗̂ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⊗̂ T̂₁) p₁ {t₁})
  | yes refl | refl = refl
∘-assoc (^ (Ŝ ⇛ T̂) (?? l)) (^ (Ŝ₂ ⊗̂ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⊗̂ T̂₁) p₁ {t₁})
  | yes refl | refl = refl
∘-assoc (^ (Ŝ₃ ⊗̂ T̂₃) p₃) (^ (Ŝ₂ ⊗̂ T̂₂) p₂ {t₂}) (^ (Ŝ₁ ⊗̂ T̂₁) p₁ {t₁})
  | yes refl | refl
  with no-gap-at-all t₂ p₃
... | refl
  with no-gap-at-all t₁ p₂
... | refl
  rewrite ∘-assoc Ŝ₃ Ŝ₂ Ŝ₁ | ∘-assoc T̂₃ T̂₂ T̂₁ = refl
∘-assoc (^ P̂₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁}) | no ¬G₁≡G₂
  with quick-gnd P̂₂ =?op quick-gnd P̂₃
∘-assoc (^ P̂₃ p₃) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁}) | no ¬G₁≡G₂ | yes refl
  with no-gap-at-all t₂ p₃
∘-assoc (^ B̂ p₃) (^ B̂ p₂ {t₂}) (^ B̂ p₁ {t₁}) | no ¬G₁≡G₂
  | yes refl | refl = ⊥-elim (¬G₁≡G₂ refl)
∘-assoc (^ B̂ p₃) (^ B̂ (?? l) {t₂}) (^ (Ŝ ⇛ T̂) p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | refl = refl
∘-assoc (^ B̂ p₃) (^ B̂ (?? l) {t₂}) (^ (Ŝ ⊗̂ T̂) p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | refl = refl
∘-assoc (^ (Ŝ₃ ⇛ T̂₃) p₃) (^ (Ŝ₂ ⇛ T̂₂) (?? l) {t₂}) (^ B̂ p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | refl = refl
∘-assoc (^ (Ŝ₃ ⇛ T̂₃) p₃) (^ (Ŝ₂ ⇛ T̂₂) p₂ {t₂}) (^ (Ŝ ⇛ T̂) p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | refl = ⊥-elim (¬G₁≡G₂ refl)
∘-assoc (^ (Ŝ₃ ⇛ T̂₃) p₃) (^ (Ŝ₂ ⇛ T̂₂) (?? l) {t₂}) (^ (Ŝ ⊗̂ T̂) p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | refl = refl
∘-assoc (^ (Ŝ₃ ⊗̂ T̂₃) p₃) (^ (Ŝ₂ ⊗̂ T̂₂) (?? l) {t₂}) (^ B̂ p₁ {t₁})
```

```
  | no ¬G₁≡G₂ | yes refl | refl = refl
∘-assoc (^ (Ŝ₃ ⊗̂ T̂₃) p₃) (^ (Ŝ₂ ⊗̂ T̂₂) (?? l) {t₂}) (^ (Ŝ ⇒̂ T̂) p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | refl = refl
∘-assoc (^ (Ŝ₃ ⊗̂ T̂₃) p₃) (^ (Ŝ₂ ⊗̂ T̂₂) p₂ {t₂}) (^ (Ŝ ⊗̂ T̂) p₁ {t₁})
  | no ¬G₁≡G₂ | yes refl | refl = ⊥-elim (¬G₁≡G₂ refl)
∘-assoc (^ P̂₃ (?? l)) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | no ¬G₂≡G₃
  with quick-gnd P̂₁ =?op quick-gnd P̂₂
... | yes G₁≡G₂' = ⊥-elim (¬G₁≡G₂ G₁≡G₂')
∘-assoc (^ P̂₃ (?? l)) (^ P̂₂ (?? l₁) {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | no ¬G₂≡G₃ | no ¬G₁≡G₂' = refl
∘-assoc (^ P̂₃ (?? l)) (^ P̂₂ ϵ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | no ¬G₂≡G₃ | no ¬G₁≡G₂' = ⊥-elim (impossible-tail t₁ ¬G₁≡G₂)
∘-assoc (^ P̂₃ ϵ) (^ P̂₂ p₂ {t₂}) (^ P̂₁ p₁ {t₁})
  | no ¬G₁≡G₂ | no ¬G₂≡G₃ = ⊥-elim (impossible-tail t₂ ¬G₂≡G₃)


data Cast : Type → Type → Set where
  _⟹_[_] : ∀ S T
    → (T̂ : LabeledType S T)
    → Cast S T


_⨟_ : ∀ {T1 T2 T3} → Cast T1 T2 → Cast T2 T3 → Cast T1 T3
(T1 ⟹ T2 [ Ŝ ]) ⨟ (.T2 ⟹ T3 [ T̂ ]) = T1 ⟹ T3 [ T̂ ∘ Ŝ ]


id^ : ∀ T → LabeledType T T
id^ * = *
id^ (‘ B) = ^ B̂ ϵ {ϵ}
id^ (‘ S ⇒ T) = ^ (id^ S ⇒̂ id^ T) ϵ {ϵ}
```

```
id^ (‘ S ⊗ T) = ^ (id^ S ⊗̂ id^ T) ϵ {ϵ}


id : ∀ T → Cast T T
id T = T ⟹ T [ id^ T ]


mutual
  ⇑ : Label → ∀ T → LabeledType T ∗
  ⇑ l ∗     = ∗
  ⇑ l (‘ B)     = ^ (B̂)              ϵ {!!}
  ⇑ l (‘ S ⇒ T) = ^ (⇓ l S ⇏ ⇑ l T) ϵ {!!}
  ⇑ l (‘ S ⊗ T) = ^ (⇑ l S ⊗̂ ⇑ l T) ϵ {!!}

  ⇓ : Label → ∀ T → LabeledType ∗ T
  ⇓ l ∗     = ∗
  ⇓ l (‘ B)     = ^ (B̂)                (?? l) {ϵ}
  ⇓ l (‘ S ⇒ T) = ^ (⇑ l S ⇏ (⇓ l T)) (?? l) {ϵ}
  ⇓ l (‘ S ⊗ T) = ^ (⇓ l S ⊗̂ (⇓ l T)) (?? l) {ϵ}

⌈_⌉’ : ∀ {T1 T2} → SrcCast T1 T2 → LabeledType T1 T2
⌈ ∗   ⟹[ l ] ∗   ⌉’ = ∗
⌈ ∗   ⟹[ l ] ‘ Q ⌉’ = ⇓ l (‘ Q)
⌈ ‘ P ⟹[ l ] ∗   ⌉’ = ⇑ l (‘ P)
⌈ ‘ P ⟹[ l ] ‘ Q ⌉’ with (‘ P) ⌣? (‘ Q)
⌈ ‘ P ⟹[ l ] ‘ Q ⌉’             | no P⌣ÌÿQ = ⊥ l _ ϵ
⌈ ‘ B       ⟹[ l ] ‘ B       ⌉’ | yes ⌣B = ^ (B̂) ϵ {ϵ}
⌈ ‘ S1 ⇒ T1 ⟹[ l ] ‘ S2 ⇒ T2 ⌉’ | yes ⌣⇒
  = ^ (⌈ S2 ⟹[ l ] S1 ⌉’ ⇏ ⌈ T1 ⟹[ l ] T2 ⌉’) ϵ {ϵ}
⌈ ‘ L1 ⊗ R1 ⟹[ l ] ‘ L2 ⊗ R2 ⌉’ | yes ⌣⊗
```

```
  = ^ (⌈ L1 ⟹[ l ] L2 ⌉' ⊛̂ ⌈ R1 ⟹[ l ] R2 ⌉') ϵ {ϵ}


⌈_⌉ : ∀ {T1 T2} → SrcCast T1 T2 → Cast T1 T2
⌈ S ⟹[ l ] T ⌉ = S ⟹ T [ ⌈ S ⟹[ l ] T ⌉' ]


open import R.BlameStrategies Label using (BlameStrategy; LazyUDBS)
open BlameStrategy LazyUDBS using (Injectable)


open import Error
  using (Error; return; raise; _>>=_; _>=>_
        ;>>=-return; >>=-assoc; >=>-assoc; >=>-cong
        ;drop-after-raise
        ;>>=-extensionality)
open import S.Values Label Injectable Cast


CastResult : Type → Set
CastResult T = Error Label (Value T)


typeop→ground : (op : TypeOp) → Ground (op · replicate *)
typeop→ground `B = `B
typeop→ground `⊗ = `⊗
typeop→ground `⇒ = `⇒


project : ∀ G {S Ts}
  → OptionalLabel G S Ts
  → Value S
  → CastResult (` G · Ts)
project H ϵ v = return v
```

```
project H (⁇ l) (dyn G v) with G =?G (typeop→ground H)
project H (⁇ l) (dyn G v) | yes refl = return v
project H (⁇ l) (dyn G v) | no ¬G≡H  = raise l


proxy : ∀ {G} Ss Ts
  → LabeledPreType G Ss Ts
  → Value (‘ G · Ss)
  ---
  → Value (‘ G · Ts)
proxy [] [] B̂ v
  = v
proxy (S₂ :: T₂ :: []) (S₃ :: T₃ :: []) (Ŝ ⇛ T̂) (lam⟨ c ⇒ d ⟩ e E)
  = lam⟨ (S₃ ⟹ S₂ [ Ŝ ]) ⨟ c ⇒ d ⨟ (T₂ ⟹ T₃ [ T̂ ]) ⟩ e E
proxy (S₂ :: T₂ :: []) (S₃ :: T₃ :: []) (Ŝ ⊗ T̂) (cons⟨ c ⊗ d ⟩ u v)
  = cons⟨ c ⨟ (S₂ ⟹ S₃ [ Ŝ ]) ⊗ d ⨟ (T₂ ⟹ T₃ [ T̂ ]) ⟩ u v


quick-left-types : ∀ S G {Ss}
  → OptionalLabel G S Ss
  → Vec Type (arity G)
quick-left-types S G {Ss} p = Ss


quick-right-types : ∀ T G {Ts}
  → Tail G Ts T
  → Vec Type (arity G)
quick-right-types T G {Ts} p = Ts


left-types : ∀ S G {Ss}
  → OptionalLabel G S Ss
```

```
  → Vec Type (arity G)
left-types ∗ G (?? l) = replicate ∗
left-types (‘ G · Ss) G ϵ = Ss


right-types : ∀ T G {Ts}
  → Tail G Ts T
  → Vec Type (arity G)
right-types ∗ G !! = replicate ∗
right-types (‘ G · Ts) G ϵ = Ts


left-types-faithful : ∀ S G {Ss}
  → (p : OptionalLabel G S Ss)
  → left-types S G p ≡ quick-left-types S G p
left-types-faithful ∗ G (?? l) = refl
left-types-faithful (‘ .(G · _)) G ϵ = refl


right-types-faithful : ∀ T G {Ts}
  → (t : Tail G Ts T)
  → right-types T G t ≡ quick-right-types T G t
right-types-faithful ∗ G !! = refl
right-types-faithful (‘ G · _) G ϵ = refl


inject : ∀ G {Ss} T
  → {t : Tail G Ss T}
  → Value (‘ G · Ss)
  → Value T
inject G ∗     {!!} v = dyn (typeop→ground G) v
inject G (‘ P) {ϵ} v = v
```

```
⟦_⟧ : ∀ {T1 T2}
  → Cast T1 T2
  → Value T1
  ---
  → CastResult T2
⟦ .* ⟹ .* [ * ] ⟧ v = return v
⟦ S ⟹ T [ ⊥ l G p ] ⟧ v = project G p v >>= λ _ → raise l
⟦ S ⟹ T [ ^ P̂ p {t} ] ⟧ v
  = project (quick-gnd P̂) p v >>= λ u
  → return (inject (quick-gnd P̂) T {t}
                   (proxy (quick-left-types S (quick-gnd P̂) p)
                          (quick-right-types T (quick-gnd P̂) t)
                          P̂ u))

⨟-assoc : ∀ {T1 T2 T3 T4}
  → (c₁ : Cast T1 T2)
  → (c₂ : Cast T2 T3)
  → (c₃ : Cast T3 T4)
  → ((c₁ ⨟ c₂) ⨟ c₃) ≡ (c₁ ⨟ (c₂ ⨟ c₃))
⨟-assoc (T₁ ⟹ T₂ [ T̂₁ ]) (T₂ ⟹ T₃ [ T̂₂ ]) (T₃ ⟹ T₄ [ T̂₃ ]) =
  begin
    ((T₁ ⟹ T₂ [ T̂₁ ]) ⨟ (T₂ ⟹ T₃ [ T̂₂ ])) ⨟ (T₃ ⟹ T₄ [ T̂₃ ])
  ≡⟨⟩
    (T₁ ⟹ T₄ [ T̂₃ ∘ (T̂₂ ∘ T̂₁) ])
  ≡⟨ cong (λ □ → T₁ ⟹ T₄ [ □ ]) (∘-assoc T̂₃ T̂₂ T̂₁) ⟩
    (T₁ ⟹ T₄ [ (T̂₃ ∘ T̂₂) ∘ T̂₁ ])
  ≡⟨⟩
```

```
    (T₁ ⟹ T₂ [ T̂₁ ]) ⨟ ((T₂ ⟹ T₃ [ T̂₂ ]) ⨟ (T₃ ⟹ T₄ [ T̂₃ ]))
    □


○-identity^l : ∀ {S} T → (T̂ : LabeledType S T) → id^ T ○ T̂ ≡ T̂
○-identity^r : ∀ S {T} → (T̂ : LabeledType S T) → T̂ ○ id^ S ≡ T̂


○-identity^l .⋆ ⋆ = refl
○-identity^l T (⊥ l G p) = refl
○-identity^l ⋆ (^ P̂ p) = dyn-left (^ P̂ p)
○-identity^l (` B) (^ B̂ p {ϵ}) = refl
○-identity^l (` S ⇒ T) (^ (Ŝ ⇛ T̂) p {ϵ}) =
  begin
    id^ (` S ⇒ T) ○ ^ (Ŝ ⇛ T̂) p {ϵ}
  ≡⟨⟩
    ^ ((Ŝ ○ id^ S) ⇛ (id^ T ○ T̂)) p {ϵ}
  ≡⟨ cong (λ □ → ^ (□ ⇛ (id^ T ○ T̂)) p {ϵ}) (○-identity^r S Ŝ) ⟩
    ^ (Ŝ ⇛ (id^ T ○ T̂)) p {ϵ}
  ≡⟨ cong (λ □ → ^ (Ŝ ⇛ □) p {ϵ}) (○-identity^l T T̂) ⟩
    ^ (Ŝ ⇛ T̂) p {ϵ}
    □
○-identity^l (` S ⊗ T) (^ (Ŝ ⊗̂ T̂) p {ϵ}) =
  begin
    id^ (` S ⊗ T) ○ ^ (Ŝ ⊗̂ T̂) p {ϵ}
  ≡⟨⟩
    ^ ((id^ S ○ Ŝ) ⊗̂ (id^ T ○ T̂)) p {ϵ}
  ≡⟨ cong (λ □ → ^ (□ ⊗̂ (id^ T ○ T̂)) p {ϵ}) (○-identity^l S Ŝ) ⟩
    ^ (Ŝ ⊗̂ (id^ T ○ T̂)) p {ϵ}
  ≡⟨ cong (λ □ → ^ (Ŝ ⊗̂ □) p {ϵ}) (○-identity^l T T̂) ⟩
```

```
    ^ (Ŝ ⊗̂ T̂) p {ϵ}
    □


∘-identity^r ∗ T̂ = refl
∘-identity^r (‘ B) (⊥ l ‘B ϵ) = refl
∘-identity^r (‘ S ⊗ T) (⊥ l ‘⊗ ϵ) = refl
∘-identity^r (‘ S ⇒ T) (⊥ l ‘⇒ ϵ) = refl
∘-identity^r (‘ B) (^ B̂ ϵ) = refl
∘-identity^r (‘ S ⇒ T) (^ (Ŝ ⇛̂ T̂) ϵ) =
  begin
    (^ (Ŝ ⇛̂ T̂) ϵ ∘ id^ (‘ S ⇒ T))
  ≡⟨⟩
    ^ ((id^ S ∘ Ŝ) ⇛̂ (T̂ ∘ id^ T)) ϵ
  ≡⟨ cong (λ □ → ^ (□ ⇛̂ (T̂ ∘ id^ T)) ϵ) (∘-identity^l S Ŝ) ⟩
    ^ (Ŝ ⇛̂ (T̂ ∘ id^ T)) ϵ
  ≡⟨ cong (λ □ → ^ (Ŝ ⇛̂ □) ϵ) (∘-identity^r T T̂) ⟩
    ^ (Ŝ ⇛̂ T̂) ϵ
    □
∘-identity^r (‘ S ⊗ T) (^ (Ŝ ⊗̂ T̂) ϵ) =
  begin
    (^ (Ŝ ⊗̂ T̂) ϵ ∘ id^ (‘ S ⊗ T))
  ≡⟨⟩
    ^ ((Ŝ ∘ id^ S) ⊗̂ (T̂ ∘ id^ T)) ϵ
  ≡⟨ cong (λ □ → ^ (□ ⊗̂ (T̂ ∘ id^ T)) ϵ) (∘-identity^r S Ŝ) ⟩
    ^ (Ŝ ⊗̂ (T̂ ∘ id^ T)) ϵ
  ≡⟨ cong (λ □ → ^ (Ŝ ⊗̂ □) ϵ) (∘-identity^r T T̂) ⟩
    ^ (Ŝ ⊗̂ T̂) ϵ
    □
```

```
⨟-identity^l : ∀ {S T} → (c : Cast S T) → id S ⨟ c ≡ c
⨟-identity^l (S ⟹ T [ T̂ ]) = cong (λ □ → S ⟹ T [ □ ]) (∘-identity^r S T̂)

⨟-identity^r : ∀ {S T} → (c : Cast S T) → c ⨟ id T ≡ c
⨟-identity^r (S ⟹ T [ T̂ ]) = cong (λ □ → S ⟹ T [ □ ]) (∘-identity^l T T̂)

lem-id : ∀ T
  → (v : Value T)
  ----------------------------
  → ⟦ id T ⟧ v ≡ return v
lem-id (*) v = refl
lem-id (` B) v = refl
lem-id (` S ⊗ T) (cons⟨ c1 ⊗ c2 ⟩ v1 v2) =
  begin
    ⟦ id (` S ⊗ T) ⟧ (cons⟨ c1 ⊗ c2 ⟩ v1 v2)
  ≡⟨⟩
    return (cons⟨ (c1 ⨟ id S) ⊗ (c2 ⨟ id T) ⟩ v1 v2)
  ≡⟨ cong (λ □ → return (cons⟨ □ ⊗ (c2 ⨟ id T) ⟩ v1 v2)) (⨟-identity^r c1)  ⟩
    return (cons⟨ c1 ⊗ (c2 ⨟ id T) ⟩ v1 v2)
  ≡⟨ cong (λ □ → return (cons⟨ c1 ⊗ □ ⟩ v1 v2)) (⨟-identity^r c2) ⟩
    return (cons⟨ c1 ⊗ c2 ⟩ v1 v2)
   □
lem-id (` S ⇒ T) (lam⟨ c1 ⇒ c2 ⟩ e E) =
  begin
    ⟦ id (` S ⇒ T) ⟧ (lam⟨ c1 ⇒ c2 ⟩ e E)
  ≡⟨⟩
    return (lam⟨ (id S ⨟ c1) ⇒ (c2 ⨟ id T) ⟩ e E)
```

```
  ≡⟨ cong (λ □ → return (lam⟨ □ ⇒ (c2 ⨟ id T) ⟩ e E)) (⨟-identity^l c1)  ⟩
    return (lam⟨ c1 ⇒ (c2 ⨟ id T) ⟩ e E)
  ≡⟨ cong (λ □ → return (lam⟨ c1 ⇒ □ ⟩ e E)) (⨟-identity^r c2) ⟩
    return (lam⟨ c1 ⇒ c2 ⟩ e E)
    □


project-inject-cancel : ∀ {G T Ss Ts}
  → (p : OptionalLabel G T Ts)
  → (t : Tail G Ss T)
  → (v : Value (‘ G · Ss))
  → ∃[ u ](project G p (inject G T {t} v) ≡ return u)
project-inject-cancel ϵ ϵ v = _ , refl
project-inject-cancel {‘B} (?? l) !! u = u , refl
project-inject-cancel {‘⊗} (?? l) !! u = u , refl
project-inject-cancel {‘⇒} (?? l) !! u = u , refl


project-inject-succeed : ∀ {G T Ts}
  → (p : OptionalLabel G T Ts)
  → (t : Tail G Ts T)
  → (v : Value (‘ G · Ts))
  → project G p (inject G T {t} v) ≡ return v
project-inject-succeed ϵ ϵ v = refl
project-inject-succeed {‘B} (?? l) !! u = refl
project-inject-succeed {‘⊗} (?? l) !! u = refl
project-inject-succeed {‘⇒} (?? l) !! u = refl


project-inject-fail : ∀ {G H Ss}
  → (¬G≡H : ¬ G ≡ H)
```

```
  → (l : Label)
  → (t : Tail G Ss ∗)
  → (v : Value (‘ G · Ss))
  → (project H (?? l) (inject G ∗ {t} v) ≡ raise l)
project-inject-fail {G} {H} ¬G≡H l ‼ v
  with (typeop→ground G) =?G (typeop→ground H)
... | yes refl = ⊥-elim (¬G≡H refl)
... | no ¬G≡H’ = refl


lem-seq : ∀ {T1 T2 T3}
  → (c1 : Cast T1 T2)
  → (c2 : Cast T2 T3)
  → (v  : Value T1)
  --------------------
  → ⟦ c1 ⨟ c2 ⟧ v ≡ ⟦ c1 ⟧ v >>= ⟦ c2 ⟧
lem-seq (.∗ ⟹ .∗ [ ∗ ]) (.∗ ⟹ T3 [ T̂2 ]) v = refl
lem-seq (T1 ⟹ T2 [ ⊥ l G p ]) (T2 ⟹ T3 [ T̂2 ]) v =
  begin
    ⟦ (T1 ⟹ T2 [ ⊥ l G p ]) ⨟ (T2 ⟹ T3 [ T̂2 ]) ⟧ v
  ≡⟨⟩
    ⟦ T1 ⟹ T3 [ ⊥ l G p ] ⟧ v
  ≡⟨⟩
    (project G p v >>= λ _ → raise l)
  ≡⟨ sym (drop-after-raise (project G p v) l ⟦ T2 ⟹ T3 [ T̂2 ] ⟧) ⟩
    (project G p v >>= λ _ → raise l) >>= ⟦ T2 ⟹ T3 [ T̂2 ] ⟧
  ≡⟨⟩
    ⟦ T1 ⟹ T2 [ ⊥ l G p ] ⟧ v >>= ⟦ T2 ⟹ T3 [ T̂2 ] ⟧
    □
```

```
lem-seq (T1 ⟹ .* [ ^ P̂ p ]) (.* ⟹ .* [ * ]) v
  = sym (>>=-return (⟦ T1 ⟹ * [ ^ P̂ p ] ⟧ v))
lem-seq (T1 ⟹ T2 [ ^ P̂₁ p₁ {t} ]) (T2 ⟹ T3 [ ⊥ l₂ G₂ p₂ ]) v
  with quick-gnd P̂₁ =?op G₂
lem-seq (T1 ⟹ T2 [ ^ P̂₁ p₁ {t} ]) (T2 ⟹ T3 [ ⊥ l₂ G₂ p₂ ]) v | yes refl
  with project G₂ p₁ v
lem-seq (T1 ⟹ T2 [ ^ P̂₁ p₁ {t} ]) (T2 ⟹ T3 [ ⊥ l₂ G₂ p₂ ]) v
  | yes refl | raise l
  = refl
lem-seq (T1 ⟹ T2 [ ^ P̂₁ p₁ {t} ]) (T2 ⟹ T3 [ ⊥ l₂ G₂ p₂ ]) v
  | yes refl
  | return v₁
  with project-inject-cancel p₂ t (proxy _ _ P̂₁ v₁)
... | x , eq
  rewrite eq = refl
lem-seq (T1 ⟹ .* [ ^ P̂₁ p₁ {t} ])
        (.* ⟹ T3 [ ⊥ l₂ G₂ (?? l) ]) v
  | no ¬p
  with project _ p₁ v
... | raise  _ = refl
... | return u
  rewrite project-inject-fail ¬p l t (proxy _ _ P̂₁ u)
  = refl
lem-seq (T1 ⟹ .(‘ G₂ · _) [ ^ P̂₁ p₁ {t} ])
        (.(‘ G₂ · _) ⟹ T3 [ ⊥ l₂ G₂ ϵ ]) v
  | no ¬p = ⊥-elim (impossible-tail t ¬p)
lem-seq (T1 ⟹ T2 [ ^ P̂₁ p₁ {t} ]) (T2 ⟹ T3 [ ^ P̂₂ p₂ ]) v
  with quick-gnd P̂₁ =?op quick-gnd P̂₂
```

```
lem-seq (T1 ⟹ .* [ ^ P̂₁ p₁ {t} ]) (.* ⟹ T3 [ ^ P̂₂ (?? l) ]) v | no ¬p
  with project _ p₁ v
... | raise l' = refl
... | return u
  rewrite project-inject-fail ¬p l t (proxy _ _ P̂₁ u)
  = refl
lem-seq (T1 ⟹ .(‘ _ · _) [ ^ P̂₁ p₁ {t} ]) (.(‘ _ · _) ⟹ T3 [ ^ P̂₂ ϵ ]) v | no ¬p = ⊥-elim (impossible ¬p)
lem-seq (T1 ⟹ T2 [ ^ P̂₁ p₁ {t} ]) (T2 ⟹ T3 [ ^ P̂₂ p₂ ]) v | yes refl
  with no-gap-at-all t p₂
lem-seq (T1 ⟹ T2 [ ^ B̂ p₁ {t} ]) (T2 ⟹ T3 [ ^ B̂ p₂ ]) v
  | yes refl | refl
  with project ‘B p₁ v
... | raise _  = refl
... | return u
  rewrite project-inject-succeed p₂ t u
  = refl
lem-seq (T1 ⟹ T2 [ ^ (Ŝ₁ ⇛ T̂₁) p₁ {t} ]) (T2 ⟹ T3 [ ^ (Ŝ₂ ⇛ T̂₂) p₂ ]) v
  | yes refl | refl
  with (project ‘⇒ p₁ v)
... | raise _  = refl
... | return (lam⟨ (_ ⟹ _ [ S ]) ⇒ (_ ⟹ _ [ T ]) ⟩ e E)
  rewrite project-inject-succeed p₂ t
          (lam⟨ (_ ⟹ _ [ S ∘ Ŝ₁ ]) ⇒ (_ ⟹ _ [ T̂₁ ∘ T ]) ⟩ e E)
  | ∘-assoc S Ŝ₁ Ŝ₂
  | ∘-assoc T̂₂ T̂₁ T
  = refl
lem-seq (T1 ⟹ T2 [ ^ (Ŝ₁ ⊗ T̂₁) p₁ {t} ]) (T2 ⟹ T3 [ ^ (Ŝ₂ ⊗ T̂₂) p₂ ]) v
  | yes refl | refl
```

```
  with (project ‘⊗ p₁ v)
... | raise _  = refl
... | return (cons⟨ (_ ⟹ _ [ S ]) ⊗ (_ ⟹ _ [ T ]) ⟩ v₁ v₂)
  rewrite project-inject-succeed p₂ t
          (cons⟨ (_ ⟹ _ [ Ŝ₁ ∘ S ]) ⊗ (_ ⟹ _ [ T̂₁ ∘ T ]) ⟩ v₁ v₂)
  | ∘-assoc Ŝ₂ Ŝ₁ S
  | ∘-assoc T̂₂ T̂₁ T
  = refl



C : CastADT Injectable
C = record
    { Cast = Cast
    ; id  = id
    ; ⌈_⌉ = ⌈_⌉
    ; _⨟̊_ = _⨟̊_
    ; ⟦_⟧ = ⟦_⟧
    ; lem-id = lem-id
    ; lem-seq = lem-seq
    }


open import S.LazyUDCastADT Label


eq-¬⌣ : ∀ {T1 T2}
  → (v : Value T1)
  → (l : Label)
  → ¬ (T1 ⌣ T2)
  ---
```

```
  → (⟦ ⌈ T1 ⟹[ l ] T2 ⌉ ⟧ v)
      ≡
    (raise l)
eq-¬⌣ {∗} {∗} v l ¬p = ⊥-elim (¬p ∗⌣∗)
eq-¬⌣ {∗} {‘ P} v l ¬p = ⊥-elim (¬p (∗⌣P P))
eq-¬⌣ {‘ P} {∗} v l ¬p = ⊥-elim (¬p (P⌣∗ P))
eq-¬⌣ {‘ P} {‘ Q} v l ¬p with (‘ P) ⌣? (‘ Q)
eq-¬⌣ {‘ P} {‘ Q} v l ¬p | yes p’ = ⊥-elim (¬p p’)
eq-¬⌣ {‘ P} {‘ Q} v l ¬p | no ¬p’ = refl

lem-rewrite-inj : (l : Label)(T : Type)
  → (⇑ l T) ≡ ⌈ T ⟹[ l ] ∗ ⌉’
lem-rewrite-inj l ∗ = refl
lem-rewrite-inj l (‘ P) = refl

lem-rewrite-proj : (l : Label)(T : Type)
  → (⇓ l T) ≡ ⌈ ∗ ⟹[ l ] T ⌉’
lem-rewrite-proj l ∗ = refl
lem-rewrite-proj l (‘ P) = refl

lem-expand-inj : (l : Label)(P : PreType)
  → (⇑ l (‘ P)) ≡ (⌈ ‘ ground P ⟹[ l ] ∗ ⌉’ ∘ ⌈ (‘ P) ⟹[ l ] ‘ ground P ⌉’)
lem-expand-inj l B = refl
lem-expand-inj l (S ⇒ T)
  rewrite lem-rewrite-proj l S | lem-rewrite-inj l T
    | ∘-identity^l ∗ ⌈ T ⟹[ l ] ∗ ⌉’
  = refl
lem-expand-inj l (S ⊗ T)
```

```
  rewrite lem-rewrite-inj l S | lem-rewrite-inj l T
    | ○-identityʳ T ⌈ T ⟹[ l ] ∗ ⌉'
    | ○-identityʳ S ⌈ S ⟹[ l ] ∗ ⌉'
    | dyn-left ⌈ S ⟹[ l ] ∗ ⌉'
    | dyn-left ⌈ T ⟹[ l ] ∗ ⌉'
  = refl


lem-expand-proj : (l : Label)(P : PreType)
  → (⇓ l (‘ P)) ≡ (⌈ ‘ ground P ⟹[ l ] ‘ P ⌉' ○ ⌈ ∗ ⟹[ l ] ‘ ground P ⌉')
lem-expand-proj l B = refl
lem-expand-proj l (S ⇒ T)
  rewrite lem-rewrite-inj l S | lem-rewrite-proj l T
    | ○-identityˡ ∗ ⌈ S ⟹[ l ] ∗ ⌉'
  = refl
lem-expand-proj l (S ⊗ T)
  rewrite lem-rewrite-proj l S | lem-rewrite-proj l T
  = refl


eq-P∗ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → ¬ Ground P
  → ⟦ ⌈ (‘ P) ⟹[ l ] ∗ ⌉ ⟧ v
      ≡
    ⟦ ⌈ (‘ P) ⟹[ l ] (‘ ground P) ⌉ ⟧ v >>= ⟦ ⌈ (‘ ground P) ⟹[ l ] ∗ ⌉ ⟧
eq-P∗ {P} v l ¬gP
  rewrite lem-expand-inj l P
  | lem-seq ⌈ (‘ P) ⟹[ l ] (‘ ground P) ⌉ ⌈ (‘ ground P) ⟹[ l ] ∗ ⌉ v
```

```
  = refl


eq-I* : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → ⟦ ⌈ ‘ P ⟹[ l ] * ⌉ ⟧ v
      ≡
    return (dyn gP v)
eq-I* {.B} v l ‘B = refl
eq-I* {.(* ⇒ *)} (lam⟨ c1 ⇒ c2 ⟩ e E) l ‘⇒
  rewrite ⨟-identity^r c2 | ⨟-identity^l c1
  = refl
eq-I* {.(* ⊗ *)} (cons⟨ c1 ⊗ c2 ⟩ v v₁) l ‘⊗
  rewrite ⨟-identity^r c1 | ⨟-identity^r c2
  = refl


eq-*P : ∀ {P}
  → (v : Value *)
  → (l : Label)
  → ¬ Ground P
  → ⟦ ⌈ * ⟹[ l ] (‘ P) ⌉ ⟧ v
      ≡
    ⟦ ⌈ * ⟹[ l ] (‘ ground P) ⌉ ⟧ v >>= ⟦ ⌈ (‘ ground P) ⟹[ l ] (‘ P) ⌉ ⟧
eq-*P {P} v l ¬gP
  rewrite lem-expand-proj l P
  | lem-seq ⌈ * ⟹[ l ] (‘ ground P) ⌉ ⌈ (‘ ground P) ⟹[ l ] (‘ P) ⌉ v
  = refl
```

```
eq-⋆I-succ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → ⟦ ⌈ ⋆ ⟹[ l ] (‘ P) ⌉ ⟧ (dyn gP v)
      ≡
    return v
eq-⋆I-succ v l ‘B = refl
eq-⋆I-succ (lam⟨ c1 ⇒ c2 ⟩ e E) l ‘⇒
  rewrite ⨟-identity^l c1 | ⨟-identity^r c2
  = refl
eq-⋆I-succ (cons⟨ c1 ⊗ c2 ⟩ v v₁) l ‘⊗
  rewrite ⨟-identity^r c1 | ⨟-identity^r c2
  = refl

eq-⋆I-fail : {P Q : PreType}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → (gQ : Ground Q)
  → ¬ (_≡_ {A = Type} (‘ P) (‘ Q))
  → ⟦ ⌈ ⋆ ⟹[ l ] (‘ Q) ⌉ ⟧ (dyn gP v)
      ≡
    raise l
eq-⋆I-fail {B} v l ‘B ‘B ¬p = ⊥-elim (¬p refl)
eq-⋆I-fail {B} v l ‘B ‘⇒ ¬p = refl
eq-⋆I-fail {B} v l ‘B ‘⊗ ¬p = refl
```

```
eq-⋆I-fail {.⋆ ⇒ .⋆} v l ‘⇒ ‘B ¬p = refl

eq-⋆I-fail {.⋆ ⇒ .⋆} v l ‘⇒ ‘⇒ ¬p = ⊥-elim (¬p refl)

eq-⋆I-fail {.⋆ ⇒ .⋆} v l ‘⇒ ‘⊗ ¬p = refl

eq-⋆I-fail {.⋆ ⊗ .⋆} v l ‘⊗ ‘B ¬p = refl

eq-⋆I-fail {.⋆ ⊗ .⋆} v l ‘⊗ ‘⇒ ¬p = refl

eq-⋆I-fail {.⋆ ⊗ .⋆} v l ‘⊗ ‘⊗ ¬p = ⊥-elim (¬p refl)


C-LazyUD : LazyUD C

C-LazyUD = record

             { eq-¬⌣ = eq-¬⌣

             ; eq-⋆⋆ = λ l v → refl

             ; eq-P⋆ = eq-P⋆

             ; eq-I⋆ = eq-I⋆

             ; eq-⋆P = eq-⋆P

             ; eq-⋆I-succ = eq-⋆I-succ

             ; eq-⋆I-fail = eq-⋆I-fail

             ; eq-B = λ l b → refl

             ; eq-⇒ = λ T21 T22 T11 T12 {S} {T} l {Γ} c₁ c₂ e E → refl

             ; eq-⊗ = λ T21 T22 T11 T12 {S} {T} l c₁ c₂ v1 v2 → refl

             }
```

D.3 Lazy UD Hypercoercions

```
module CastRepresentations.LazyUDHypercoercions (Label : Set) where


open import Types

  renaming (B to TB; _⇒_ to _T⇒_; _⊗_ to _T⊗_)

open import Cast Label using (_⟹[_]_) renaming (Cast to SrcCast)

open import Terms Label
```

```
open import S.CastADT Label


open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Data.Empty using (⊥-elim)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Data.Product using (_×_; ∃-syntax; _,_)
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; sym; cong; cong₂)


infix 100 _⇒_
infix 100 _⊗_


data Head : Type → PreType → Set where
  ϵ : ∀ {P}
      ---
    → Head (‘ P) P
  ?? : ∀ {P}
    → (gP : Ground P)
    → (l : Label)
      ---
    → Head ∗ P


data Tail : PreType → Type → Set where
  ϵ : ∀ {P}
      ------------
    → Tail P (‘ P)


  !! : ∀ {P}
```

```
    → (gP : Ground P)
      ---------------
    → Tail P *


mutual
  data Cast : Type → Type → Set where
    id* :
      ---
      Cast * *
    ↷ : ∀ {A P Q B} →
      (h : Head A P) →
      (m : Body P Q) →
      (t : Tail Q B) →
      ---
      Cast A B


  data Body : PreType → PreType → Set where
    ⊥ : ∀ {P Q}
      → (l : Label)
        ---
      → Body P Q


    ‘_ : ∀ {P Q} →
      (m : PreBody P Q) →
      ---
      Body P Q


  data PreBody : PreType → PreType → Set where
```

```
    B :
      ---
      PreBody TB TB
    _⇒_ : ∀ {S1 S2 T1 T2} →
      (c₁ : Cast S2 S1) →
      (c₂ : Cast T1 T2) →
      ---
      PreBody (S1 T⇒ T1) (S2 T⇒ T2)
    _⊗_ : ∀ {S1 S2 T1 T2} →
      (c₁ : Cast S1 S2) →
      (c₂ : Cast T1 T2) →
      ---
      PreBody (S1 T⊗ T1) (S2 T⊗ T2)


data CompatibleTailHead {P : PreType} : ∀ {T} → Tail P T → Head T P → Set where
  none : CompatibleTailHead ϵ ϵ
  some : ∀ {l}
    → (G : Ground P)
    → CompatibleTailHead (!! G) (?? G l)


data Gap : ∀ {P T Q} → Tail P T → Head T Q → Set where
  some : ∀ {P Q}
    → {gP : Ground P}
    → {gQ : Ground Q}
    → (¬p : ¬ (P ≡ Q))
    → (l  : Label)
    → Gap (!! gP) (?? gQ l)
```

```
  none : ∀ {P T}
    → {t : Tail P T}
    → {h : Head T P}
    → (p : CompatibleTailHead t h)
    → Gap t h


check-gap : ∀ {P T Q}
  → (t : Tail P T)
  → (h : Head T Q)
  → Gap t h
check-gap ϵ ϵ = none none
check-gap (!! gP) (?? gQ l) with gP =?G gQ
check-gap (!! gP) (?? gQ l) | yes refl rewrite ground-unique gP gQ = none (some gQ)
check-gap (!! gP) (?? gQ l) | no ¬P≡Q  = some ¬P≡Q l


mutual
  _⨟_ : ∀ {T1 T2 T3} → Cast T1 T2 → Cast T2 T3 → Cast T1 T3
  id*        ⨟ d   = d
  ↷ h1 m1 t1 ⨟ id* = ↷ h1 m1 t1
  ↷ h1 m1 t1 ⨟ ↷ h2 m2 t2 = ↷ h1 (seq-m m1 t1 h2 m2) t2


  seq-m : ∀ {P1 P2 T P3 P4}
    → Body P1 P2
    → Tail P2 T
    → Head T  P3
    → Body P3 P4
    → Body P1 P4
  seq-m (⊥ l1) t1 h2 m2 = ⊥ l1
```

```
  seq-m (‘ m1) t1 h2 m2 with check-gap t1 h2
  seq-m (‘ m1) .(!! _) .(?? _ l) m2 | some ¬P≡Q l = ⊥ l
  seq-m (‘ m1) t1 h2 (⊥ l2) | none p = ⊥ l2
  seq-m (‘ m1) t1 h2 (‘ m2) | none p = ‘ (m1 ⨟’ m2)

  _⨟’_ : ∀ {T1 T2 T3} → PreBody T1 T2 → PreBody T2 T3 → PreBody T1 T3
  B ⨟’ B = B
  (c₁ ⇒ c₂) ⨟’ (c₃ ⇒ c₄) = (c₃ ⨟ c₁) ⇒ (c₂ ⨟ c₄)
  (c₁ ⊗ c₂) ⨟’ (c₃ ⊗ c₄) = (c₁ ⨟ c₃) ⊗ (c₂ ⨟ c₄)

mutual
  ⇑ : Label → ∀ T → Cast T ⋆
  ⇑ l ⋆     = id⋆
  ⇑ l (‘ TB)     = ↷ ϵ (‘ B)                (!! ‘B)
  ⇑ l (‘ S T⇒ T) = ↷ ϵ (‘ (⇓ l S ⇒ ⇑ l T)) (!! ‘⇒)
  ⇑ l (‘ S T⊗ T) = ↷ ϵ (‘ (⇑ l S ⊗ ⇑ l T)) (!! ‘⊗)

  ⇓ : Label → ∀ T → Cast ⋆ T
  ⇓ l ⋆     = id⋆
  ⇓ l (‘ TB)     = ↷ (?? ‘B l) (‘ B)                 ϵ
  ⇓ l (‘ S T⇒ T) = ↷ (?? ‘⇒ l) (‘ ⇑ l S ⇒ (⇓ l T)) ϵ
  ⇓ l (‘ S T⊗ T) = ↷ (?? ‘⊗ l) (‘ ⇓ l S ⊗ (⇓ l T)) ϵ

⌈_⌉ : ∀ {T1 T2} → SrcCast T1 T2 → Cast T1 T2
⌈ ⋆   ⟹[ l ] ⋆   ⌉ = id⋆
⌈ ⋆   ⟹[ l ] ‘ Q ⌉ = ⇓ l (‘ Q)
⌈ ‘ P ⟹[ l ] ⋆   ⌉ = ⇑ l (‘ P)
⌈ ‘ P ⟹[ l ] ‘ Q ⌉ with (‘ P) ⌣? (‘ Q)
```

```
⌈ ‘ P ⟹[ l ] ‘ Q ⌉                | no P⌣ÌÿQ = ⌒↷ ϵ (⊥ l) ϵ
⌈ ‘ TB       ⟹[ l ] ‘ TB       ⌉ | yes ⌣B = ⌒↷ ϵ (‘ B) ϵ
⌈ ‘ S1 T⇒ T1 ⟹[ l ] ‘ S2 T⇒ T2 ⌉ | yes ⌣⇒
  = ⌒↷ ϵ (‘ ⌈ S2 ⟹[ l ] S1 ⌉ ⇒ ⌈ T1 ⟹[ l ] T2 ⌉) ϵ
⌈ ‘ L1 T⊗ R1 ⟹[ l ] ‘ L2 T⊗ R2 ⌉ | yes ⌣⊗
  = ⌒↷ ϵ (‘ ⌈ L1 ⟹[ l ] L2 ⌉ ⊗ ⌈ R1 ⟹[ l ] R2 ⌉) ϵ


mutual
  id : ∀ T → Cast T T
  id *
    = id*
  id (‘ P)
    = ⌒↷ ϵ (‘ id-m P) ϵ


  id-m : ∀ P → PreBody P P
  id-m TB
    = B
  id-m (S T⇒ T)
    = id S ⇒ id T
  id-m (L T⊗ R)
    = (id L) ⊗ (id R)


open import R.BlameStrategies Label using (BlameStrategy; LazyUDBS)
open BlameStrategy LazyUDBS using (Injectable)


open import Error
  using (Error; return; raise; _>>=_; _>=>_
        ;>>=-return; >>=-assoc; >=>-assoc; >=>-cong)
```

```
open import S.Values Label Injectable Cast


CastResult : Type → Set
CastResult T = Error Label (Value T)


⟦_⟧t : ∀ {P T}
  → Tail P T
  → Value (‘ P)
  ---
  → CastResult T
⟦ !! gP ⟧t v = return (dyn gP v)
⟦ ϵ    ⟧t v = return v


proxy : ∀ {P1 P2}
  → Value (‘ P1)
  → PreBody P1 P2
  ---
  → Value (‘ P2)
proxy v B = v
proxy (lam⟨ c1 ⇒ c2 ⟩ e E) (c3 ⇒ c4)    = lam⟨ c3 ⨟ c1 ⇒ c2 ⨟ c4 ⟩ e E
proxy (cons⟨ c1 ⊗ c2 ⟩ v1 v2) (c3 ⊗ c4) = cons⟨ c1 ⨟ c3 ⊗ c2 ⨟ c4 ⟩ v1 v2


⟦_⟧m : ∀ {P1 P2}
  → Body P1 P2
  → Value (‘ P1)
  → CastResult (‘ P2)
⟦ ⊥ l ⟧m v = raise l
⟦ ‘ m ⟧m v = return (proxy v m)
```

```
⟦_⟧h : ∀ {T P}
  → Head T P
  → Value T
  → CastResult (‘ P)
⟦ ϵ      ⟧h v = return v
⟦ ?? gQ l ⟧h (dyn gP v) with gP =?G gQ
⟦ ?? gQ l ⟧h (dyn gP v) | yes refl = return v
⟦ ?? gQ l ⟧h (dyn gP v) | no ¬P≡Q  = raise l

⟦_⟧ : ∀ {T1 T2}
  → Cast T1 T2
  → Value T1
  ---
  → CastResult T2
⟦ id*     ⟧ v = return v
⟦ ↷ h m t ⟧ v = ⟦ h ⟧h v >>= ⟦ m ⟧m >>= ⟦ t ⟧t

mutual
  identity^l : ∀ {T1 T2} → (c : Cast T1 T2) → id T1 ⨟ c ≡ c
  identity^l id* = refl
  identity^l (↷ (?? P l) m t2) = refl
  identity^l (↷ ϵ (⊥ l) t2) = refl
  identity^l (↷ ϵ (‘ B) t2) = refl
  identity^l (↷ ϵ (‘ (c₁ ⇒ c₂)) t2) rewrite identity^r c₁ | identity^l c₂ = refl
  identity^l (↷ ϵ (‘ (c₁ ⊗ c₂)) t2) rewrite identity^l c₁ | identity^l c₂ = refl

  identity^r : ∀ {T1 T2} → (c : Cast T1 T2) → c ⨟ id T2 ≡ c
```

```
  identity^r id* = refl

  identity^r (↷ t1 m (!! P)) = refl

  identity^r (↷ t1 (⊥ l) ε) = refl

  identity^r (↷ t1 (‘ B) ε) = refl

  identity^r (↷ t1 (‘ (c₁ ⇒ c₂)) ε) rewrite identity^l c₁ | identity^r c₂ = refl

  identity^r (↷ t1 (‘ (c₁ ⊗ c₂)) ε) rewrite identity^r c₁ | identity^r c₂ = refl


mutual

  assoc : ∀ {T1 T2 T3 T4}

    → (c1 : Cast T1 T2)

    → (c2 : Cast T2 T3)

    → (c3 : Cast T3 T4)

    → (c1 ⨟ c2) ⨟ c3 ≡ c1 ⨟ (c2 ⨟ c3)

  assoc id* c2 c3 rewrite identity^l c2 | identity^l (c2 ⨟ c3) = refl

  assoc (↷ h₁ m₁ t₁) id* c3 rewrite identity^l c3 = refl

  assoc (↷ h₁ m₁ t₁) (↷ h₂ m₂ t₂) id* = refl

  assoc (↷ h₁ m₁ t₁) (↷ h₂ m₂ t₂) (↷ h₃ m₃ t₃)

    = cong (λ □ → ↷ h₁ □ t₃) (assoc-seq-m m₁ t₁ h₂ m₂ t₂ h₃ m₃)


  assoc-seq-m : ∀ {P1 P2 T1 P3 P4 T2 P5 P6}

    → (m1 : Body P1 P2)

    → (t1 : Tail P2 T1)

    → (h2 : Head T1 P3)

    → (m2 : Body P3 P4)

    → (t2 : Tail P4 T2)

    → (h3 : Head T2 P5)

    → (m3 : Body P5 P6)

    → (seq-m (seq-m m1 t1 h2 m2) t2 h3 m3) ≡ (seq-m m1 t1 h2 (seq-m m2 t2 h3 m3))
```

```
  assoc-seq-m (⊥ l1) t1 h2 m2 t2 h3 m3 = refl
  assoc-seq-m (‘ m1) t1 h2 m2 t2 h3 m3 with check-gap t1 h2
  assoc-seq-m (‘ m1) t1 h2 m2 t2 h3 m3 | some ¬P≡Q l = refl
  assoc-seq-m (‘ m1) t1 h2 (⊥ l2) t2 h3 m3 | none p = refl
  assoc-seq-m (‘ m1) t1 h2 (‘ m2) t2 h3 m3 | none p with check-gap t2 h3
  assoc-seq-m (‘ m1) t1 h2 (‘ m2) t2 h3 m3 | none p | some ¬P≡Q l = refl
  assoc-seq-m (‘ m1) t1 h2 (‘ m2) t2 h3 (⊥ l3) | none p | none q = refl
  assoc-seq-m (‘ B) t1 h2 (‘ B) t2 h3 (‘ B) | none p | none q = refl
  assoc-seq-m (‘ (c₁ ⇒ c₂)) t1 h2 (‘ (c₃ ⇒ c₄)) t2 h3 (‘ (c₅ ⇒ c₆))
    | none p | none q
    = cong₂ (λ □ ■ → (‘ □ ⇒ ■)) (sym (assoc c₅ c₃ c₁)) (assoc c₂ c₄ c₆)
  assoc-seq-m (‘ (c₁ ⊗ c₂)) t1 h2 (‘ (c₃ ⊗ c₄)) t2 h3 (‘ (c₅ ⊗ c₆))
    | none p | none q
    = cong₂ (λ □ ■ → (‘ □ ⊗ ■)) (assoc c₁ c₃ c₅) (assoc c₂ c₄ c₆)


lem-id-m : ∀ {P}
  → (v : Value (‘ P))
  ----------------------------
  → proxy v (id-m P) ≡ v
lem-id-m {TB} v = refl
lem-id-m {S T⇒ T} (lam⟨ c ⇒ d ⟩ e E)  rewrite identityˡ c | identityʳ d = refl
lem-id-m {S T⊗ T} (cons⟨ c ⊗ d ⟩ v u) rewrite identityʳ c | identityʳ d = refl


lem-id : ∀ {T}
  → (v : Value T)
  ----------------------------
  → ⟦ id T ⟧ v ≡ return v
lem-id {∗} v = refl
```

```
lem-id {‘ P} v rewrite lem-id-m v = refl


lem-proxy : ∀ {P1 P2 P3}
  → (v : Value (‘ P1))
  → (m1 : PreBody P1 P2)
  → (m2 : PreBody P2 P3)
  → proxy v (m1 ⨟’ m2) ≡ proxy (proxy v m1) m2
lem-proxy v B B = refl
lem-proxy (lam⟨ c1 ⇒ d1 ⟩ e E) (c2 ⇒ d2) (c3 ⇒ d3)
  rewrite assoc c3 c2 c1 | assoc d1 d2 d3
  = refl
lem-proxy (cons⟨ c1 ⊗ d1 ⟩ v1 v2) (c2 ⊗ d2) (c3 ⊗ d3)
  rewrite assoc c1 c2 c3 | assoc d1 d2 d3
  = refl


lem-seq-m : ∀ {P1 P2 T P3 P4}
  → (m1 : Body P1 P2)
  → (t1 : Tail P2 T)
  → (h2 : Head T  P3)
  → (m2 : Body P3 P4)
  → (∀ v →
       ⟦ seq-m m1 t1 h2 m2 ⟧m v
         ≡
       (⟦ m1 ⟧m >=> ⟦ t1 ⟧t >=> ⟦ h2 ⟧h >=> ⟦ m2 ⟧m) v)
lem-seq-m (⊥ l1) t1 h2 m2 v = refl
lem-seq-m (‘ m1) ϵ ϵ (⊥ l2) v = refl
lem-seq-m (‘ m1) ϵ ϵ (‘ m2) v = cong return (lem-proxy v m1 m2)
lem-seq-m (‘ m1) (!! gP) (?? gQ l) m2 v with gP =?G gQ
```

```
lem-seq-m (‘ m1) (!! gP) (?? gQ l) m2 v | no ¬p = refl
lem-seq-m (‘ m1) (!! gP) (?? gQ l) (⊥ l2) v | yes refl
  rewrite ground-unique gP gQ = refl
lem-seq-m (‘ m1) (!! gP) (?? gQ l) (‘ m2) v | yes refl
  rewrite ground-unique gP gQ = cong return (lem-proxy v m1 m2)


lem-seq : ∀ {T1 T2 T3}
  → (c1 : Cast T1 T2)
  → (c2 : Cast T2 T3)
  → ∀ v
  --------------------
  → ⟦ c1 ⨟ c2 ⟧ v ≡ ⟦ c1 ⟧ v >>= ⟦ c2 ⟧
lem-seq id∗ c2 v = refl
lem-seq (↷ h1 m1 t1) id∗ v = sym (>>=-return _)
lem-seq (↷ h1 m1 t1) (↷ h2 m2 t2) v with ⟦ h1 ⟧h v
lem-seq (↷ h1 m1 t1) (↷ h2 m2 t2) v | raise l   = refl
lem-seq (↷ h1 m1 t1) (↷ h2 m2 t2) v | return v’
  rewrite lem-seq-m m1 t1 h2 m2 v’
  with ⟦ m1 ⟧m v’ >>= ⟦ t1 ⟧t
... | raise l    = refl
... | return v’’ = refl


H : CastADT Injectable
H = record
    { Cast = Cast
    ; id  = id
    ; ⌈_⌉ = ⌈_⌉
    ; _⨟_ = _⨟_
```

```
    ; ⟦_⟧ = ⟦_⟧
    ; lem-id = λ T v → lem-id v
    ; lem-seq = lem-seq
    }

open import S.LazyUDCastADT Label

eq-¬∪ : ∀ {T1 T2}
  → (v : Value T1)
  → (l : Label)
  → ¬ (T1 ∪ T2)
  ---
  → ⟦ ⌈ T1 ⟹[ l ] T2 ⌉ ⟧ v
      ≡
    raise l
eq-¬∪ {*} {*} v l ¬p = ⊥-elim (¬p *∪*)
eq-¬∪ {*} {‘ P} v l ¬p = ⊥-elim (¬p (*∪P P))
eq-¬∪ {‘ P} {*} v l ¬p = ⊥-elim (¬p (P∪* P))
eq-¬∪ {‘ P} {‘ Q} v l ¬p with (‘ P) ∪? (‘ Q)
eq-¬∪ {‘ P} {‘ Q} v l ¬p | yes p’ = ⊥-elim (¬p p’)
eq-¬∪ {‘ P} {‘ Q} v l ¬p | no ¬p’ = refl

lem-rewrite-inj : (l : Label)(T : Type)
  → (⇑ l T) ≡ ⌈ T ⟹[ l ] * ⌉
lem-rewrite-inj l * = refl
lem-rewrite-inj l (‘ P) = refl

lem-rewrite-proj : (l : Label)(T : Type)
```

```
  → (⇓ l T) ≡ ⌈ ∗ ⟹[ l ] T ⌉

lem-rewrite-proj l ∗ = refl

lem-rewrite-proj l (‘ P) = refl


lem-expand-inj : (l : Label)(P : PreType)

  → (⇑ l (‘ P)) ≡ (⌈ (‘ P) ⟹[ l ] ‘ ground P ⌉ ⨟ ⌈ ‘ ground P ⟹[ l ] ∗ ⌉)

lem-expand-inj l TB = refl

lem-expand-inj l (S T⇒ T)

  rewrite lem-rewrite-proj l S | lem-rewrite-inj l T

    | identityʳ ⌈ T ⟹[ l ] ∗ ⌉

  = refl

lem-expand-inj l (S T⊗ T)

  rewrite lem-rewrite-inj l S | lem-rewrite-inj l T

    | identityʳ ⌈ T ⟹[ l ] ∗ ⌉

    | identityʳ ⌈ S ⟹[ l ] ∗ ⌉

  = refl


lem-expand-proj : (l : Label)(P : PreType)

  → (⇓ l (‘ P)) ≡ (⌈ ∗ ⟹[ l ] ‘ ground P ⌉ ⨟ ⌈ ‘ ground P ⟹[ l ] ‘ P ⌉)

lem-expand-proj l TB = refl

lem-expand-proj l (S T⇒ T)

  rewrite lem-rewrite-inj l S | lem-rewrite-proj l T

    | identityʳ ⌈ S ⟹[ l ] ∗ ⌉

  = refl

lem-expand-proj l (S T⊗ T)

  rewrite lem-rewrite-proj l S | lem-rewrite-proj l T

  = refl
```

```
eq-P∗ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → ¬ Ground P
  → ⟦ ⌈ (‘ P) ⟹[ l ] ∗ ⌉ ⟧ v
      ≡
    ⟦ ⌈ (‘ P) ⟹[ l ] (‘ ground P) ⌉ ⟧ v >>= ⟦ ⌈ (‘ ground P) ⟹[ l ] ∗ ⌉ ⟧
eq-P∗ {P} v l ¬gP
  rewrite lem-expand-inj l P
  | lem-seq ⌈ (‘ P) ⟹[ l ] (‘ ground P) ⌉ ⌈ (‘ ground P) ⟹[ l ] ∗ ⌉ v
  = refl


eq-I∗ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → ⟦ ⌈ ‘ P ⟹[ l ] ∗ ⌉ ⟧ v
      ≡
    return (dyn gP v)
eq-I∗ {.TB} v l ‘B = refl
eq-I∗ {.(∗ T⇒ ∗)} (lam⟨ c1 ⇒ c2 ⟩ e E) l ‘⇒
  rewrite identity^r c2
  = refl
eq-I∗ {.(∗ T⊗ ∗)} (cons⟨ c1 ⊗ c2 ⟩ v v₁) l ‘⊗
  rewrite identity^r c1 | identity^r c2
  = refl


eq-∗P : ∀ {P}
```

```
  → (v : Value ∗)
  → (l : Label)
  → ¬ Ground P
  → ⟦ ⌈ ∗ ⟹[ l ] (‘ P) ⌉ ⟧ v
      ≡
    ⟦ ⌈ ∗ ⟹[ l ] (‘ ground P) ⌉ ⟧ v >>= ⟦ ⌈ (‘ ground P) ⟹[ l ] (‘ P) ⌉ ⟧
eq-∗P {P} v l ¬gP
  rewrite lem-expand-proj l P
  | lem-seq ⌈ ∗ ⟹[ l ] (‘ ground P) ⌉ ⌈ (‘ ground P) ⟹[ l ] (‘ P) ⌉ v
  = refl


eq-∗I-succ : ∀ {P}
  → (v : Value (‘ P))
  → (l : Label)
  → (gP : Ground P)
  → ⟦ ⌈ ∗ ⟹[ l ] (‘ P) ⌉ ⟧ (dyn gP v)
      ≡
    return v
eq-∗I-succ v l ‘B = refl
eq-∗I-succ (lam⟨ c1 ⇒ c2 ⟩ e E) l ‘⇒
  rewrite identity′ c2
  = refl
eq-∗I-succ (cons⟨ c1 ⊗ c2 ⟩ v v₁) l ‘⊗
  rewrite identity′ c1 | identity′ c2
  = refl


eq-∗I-fail : {P Q : PreType}
  → (v : Value (‘ P))
```

```
  → (l : Label)
  → (gP : Ground P)
  → (gQ : Ground Q)
  → ¬ (_≡_ {A = Type} (‘ P) (‘ Q))
  → ⟦ ⌈ * ⟹[ l ] (‘ Q) ⌉ ⟧ (dyn gP v)
      ≡
    raise l
eq-∗I-fail {TB} v l ‘B ‘B ¬p = ⊥-elim (¬p refl)
eq-∗I-fail {TB} v l ‘B ‘⇒ ¬p = refl
eq-∗I-fail {TB} v l ‘B ‘⊗ ¬p = refl
eq-∗I-fail {.∗ T⇒ .∗} v l ‘⇒ ‘B ¬p = refl
eq-∗I-fail {.∗ T⇒ .∗} v l ‘⇒ ‘⇒ ¬p = ⊥-elim (¬p refl)
eq-∗I-fail {.∗ T⇒ .∗} v l ‘⇒ ‘⊗ ¬p = refl
eq-∗I-fail {.∗ T⊗ .∗} v l ‘⊗ ‘B ¬p = refl
eq-∗I-fail {.∗ T⊗ .∗} v l ‘⊗ ‘⇒ ¬p = refl
eq-∗I-fail {.∗ T⊗ .∗} v l ‘⊗ ‘⊗ ¬p = ⊥-elim (¬p refl)


H-LazyUD : LazyUD H
H-LazyUD = record
             { eq-¬⌣ = eq-¬⌣
             ; eq-∗∗ = λ l v → refl
             ; eq-P∗ = eq-P∗
             ; eq-I∗ = eq-I∗
             ; eq-∗P = eq-∗P
             ; eq-∗I-succ = eq-∗I-succ
             ; eq-∗I-fail = eq-∗I-fail
             ; eq-B = λ l b → refl
             ; eq-⇒ = λ T21 T22 T11 T12 {S} {T} l {Γ} c₁ c₂ e E → refl
```

```
            ; eq-⊗ = λ T21 T22 T11 T12 {S} {T} l c₁ c₂ v1 v2 → refl
            }
```

D.4 Lazy D Hypercoercions

```
module CastRepresentations.LazyDHypercoercions (Label : Set) where


open import Types
open import Variables
open import Cast Label using (_⟹[_]_) renaming (Cast to SrcCast)
open import Terms Label
open import S.CastADT Label


open import Relation.Nullary using (Dec; yes; no; ¬_)
open import Data.Empty using (⊥-elim)
open import Relation.Binary.PropositionalEquality using (_≡_; refl; sym; cong; cong₂)



data Head : Type → PreType → Set where
  ?? : ∀ P →
    (l : Label) →
    ---
    Head ∗ P
  ϵ : ∀ {P} →
    ---
    Head (‘ P) P


data Tail : PreType → Type → Set where
  !! : ∀ P
```

```
      ---
    → Tail P ∗
  ϵ : ∀ {P} →
    ---
    Tail P (‘ P)



data Cast : Type → Type → Set
data Body : PreType → PreType → Set
data PreBody : PreType → PreType → Set


data Cast where
  id∗ :
    ---
    Cast ∗ ∗
  ↷ : ∀ {A P Q B} →
    (h : Head A P) →
    (m : Body P Q) →
    (t : Tail Q B) →
    ---
    Cast A B


data Body where
  ⊥ : ∀ {P Q}
    → (l : Label)
      ---
    → Body P Q
```

```
  ‘_ : ∀ {P Q} →
     (m : PreBody P Q) →
     ---
     Body P Q



data PreBody where

  B̂ : PreBody B B

  _⇒̂_ : ∀ {S1 S2 T1 T2}
    → (c₁ : Cast S2 S1)
    → (c₂ : Cast T1 T2)
      ---------------------------
    → PreBody (S1 ⇒ T1) (S2 ⇒ T2)

  _⊗̂_ : ∀ {S1 S2 T1 T2}
    → (c₁ : Cast S1 S2)
    → (c₂ : Cast T1 T2)
      ---------------------------
    → PreBody (S1 ⊗ T1) (S2 ⊗ T2)



data GapT : Type → Type → Set where
  none : ∀ {T} → GapT T T
  some : (l : Label) → ∀ T1 T2 → GapT T1 T2

GapP : PreType → PreType → Set
```

```
GapP P Q = GapT (‘ P) (‘ Q)


data Check : ∀ {P Q} → GapP P Q → Set where
  bad : ∀ {P Q}
    → (¬P⌣Q : ¬ ((‘ P) ⌣ (‘ Q)))
    → (l    : Label)
    → Check (some l (‘ P) (‘ Q))

  good : ∀ {P Q}
    → (P⌣Q : (‘ P) ⌣ (‘ Q))
    → {g : GapP P Q}
    → Check g

check-gap : ∀ {P Q}
  → (g : GapP P Q)
  → Check g
check-gap none = good (⌣refl _)
check-gap (some l (‘ P) (‘ Q)) with (‘ P) ⌣? (‘ Q)
check-gap (some l P Q) | no ¬p = bad ¬p l
check-gap (some l P Q) | yes p = good p


g-dom : ∀ {T1 T2 T3 T4}
  → GapP (T1 ⇒ T2) (T3 ⇒ T4)
  → GapT T3 T1
g-dom none = none
g-dom (some l (‘ T1 ⇒ T2) (‘ T3 ⇒ T4)) = some l T3 T1


g-cod : ∀ {T1 T2 T3 T4}
```

```
  → GapP (T1 ⇒ T2) (T3 ⇒ T4)
  → GapT T2 T4
g-cod none = none
g-cod (some l (‘ T1 ⇒ T2) (‘ T3 ⇒ T4)) = some l T2 T4


g-car : ∀ {T1 T2 T3 T4}
  → GapP (T1 ⊗ T2) (T3 ⊗ T4)
  → GapT T1 T3
g-car none = none
g-car (some l (‘ T1 ⊗ T2) (‘ T3 ⊗ T4)) = some l T1 T3


g-cdr : ∀ {T1 T2 T3 T4}
  → GapP (T1 ⊗ T2) (T3 ⊗ T4)
  → GapT T2 T4
g-cdr none = none
g-cdr (some l (‘ T1 ⊗ T2) (‘ T3 ⊗ T4)) = some l T2 T4


mk-proj : ∀ {T P}
  → GapT ∗ T
  → Head T P
  ---
  → Head ∗ P
mk-proj g (?? P l) = (?? P l)
mk-proj (some l ∗ (‘ P)) ϵ = ?? P l


mk-inj : ∀ {T P}
  → Tail P T
  → GapT T ∗
```

```
  ---
  → Tail P ∗
mk-inj (!! P) g = !! P
mk-inj ϵ (some l (‘ P) ∗) = !! P


mk-inj-none : ∀ {P}
  → (t : Tail P ∗)
  ---
  → mk-inj t none ≡ t
mk-inj-none (!! P) = refl


mk-gap : ∀ {P S T Q}
  → Tail P S
  → GapT S T
  → Head T Q
  -----------------
  → GapP P Q
mk-gap ϵ g ϵ = g
mk-gap ϵ (some l’ (‘ P) ∗)    (?? Q l) = some l (‘ P) (‘ Q)
mk-gap (!! P) g                (?? Q l) = some l (‘ P) (‘ Q)
mk-gap (!! P) (some l ∗ (‘ Q)) ϵ       = some l (‘ P) (‘ Q)


mutual
  seq : ∀ {T1 T2 T3 T4}
    → Cast T1 T2
    → GapT T2 T3
    → Cast T3 T4
  ----------------
```

```
  → Cast T1 T4
seq id∗ g id∗ = id∗
seq id∗ g (↷ h2 m2 t2) = ↷ (mk-proj g h2) m2 t2
seq (↷ h1 m1 t1) g id∗ = ↷ h1 m1 (mk-inj t1 g)
seq (↷ h1 m1 t1) g (↷ h2 m2 t2) = ↷ h1 (seq-m m1 (mk-gap t1 g h2) m2) t2


seq-m : ∀ {P1 P2 P3 P4}
  → Body P1 P2
  → GapP P2 P3
  → Body P3 P4
  ---
  → Body P1 P4
seq-m (⊥ l1) g m2 = ⊥ l1
seq-m (‘ m1) g m2 with check-gap g
seq-m (‘ m1) .(some l _ _) m2 | bad ¬P⌣Q l = ⊥ l
seq-m (‘ m1) g (⊥ l2) | good P⌣Q = ⊥ l2
seq-m (‘ m1) g (‘ m2) | good P⌣Q = ‘ seq-mm P⌣Q m1 g m2


seq-mm : ∀ {P1 P2 P3 P4}
  → (‘ P2) ⌣ (‘ P3)
  → PreBody P1 P2
  → GapP P2 P3
  → PreBody P3 P4
  ---
  → PreBody P1 P4
seq-mm ⌣B B̂ g B̂ = B̂
seq-mm ⌣⇒ (c₁ ⇏ c₂) g (c₃ ⇏ c₄) = (seq c₃ (g-dom g) c₁) ⇏ (seq c₂ (g-cod g) c₄)
seq-mm ⌣⊗ (c₁ ⊗̂ c₂) g (c₃ ⊗̂ c₄) = (seq c₁ (g-car g) c₃) ⊗̂ (seq c₂ (g-cdr g) c₄)
```

```
mutual
  id : ∀ T → Cast T T
  id ∗
    = id∗
  id (‘ P)
    = ↷ ϵ (‘ id-m P) ϵ

  id-m : ∀ P → PreBody P P
  id-m B
    = B̂
  id-m (T₁ ⇒ T₂)
    = id T₁ ⇏ id T₂
  id-m (T₁ ⊗ T₂)
    = id T₁ ⊗̂ id T₂

⌈_⌉ : ∀ {T1 T2} → SrcCast T1 T2 → Cast T1 T2
⌈ T1 ⟹[ l ] T2 ⌉ = seq (id T1) (some l T1 T2) (id T2)

_⨟_ : ∀ {T1 T2 T3} → Cast T1 T2 → Cast T2 T3 → Cast T1 T3
_⨟_ c1 c2 = seq c1 none c2

mutual
  identityˡ : ∀ {T1 T2} → (c : Cast T1 T2) → id T1 ⨟ c ≡ c
  identityˡ id∗ = refl
  identityˡ (↷ (?? P l) m t2) = refl
  identityˡ (↷ ϵ (⊥ l) t2) = refl
  identityˡ (↷ ϵ (‘ B̂) t2) = refl
```

```
  identity^l (↷ ϵ (‘ (c₁ ⇛ c₂)) t2) rewrite identity^r c₁ | identity^l c₂ = refl
  identity^l (↷ ϵ (‘ (c₁ ⊗̂ c₂)) t2) rewrite identity^l c₁ | identity^l c₂ = refl


  identity^r : ∀ {T1 T2} → (c : Cast T1 T2) → c ⨟ id T2 ≡ c
  identity^r id* = refl
  identity^r (↷ t1 m (‼ P)) = refl
  identity^r (↷ t1 (⊥ l) ϵ) = refl
  identity^r (↷ t1 (‘ B̂) ϵ) = refl
  identity^r (↷ t1 (‘ (c₁ ⇛ c₂)) ϵ) rewrite identity^l c₁ | identity^r c₂ = refl
  identity^r (↷ t1 (‘ (c₁ ⊗̂ c₂)) ϵ) rewrite identity^r c₁ | identity^r c₂ = refl


lem-mk-gap : ∀ {P1 T1 T2 P2}
  → (t1 : Tail P1 T1)
  → (g1 : GapT T1 ∗)
  → (g2 : GapT ∗  T2)
  → (h2 : Head T2 P2)
  → mk-gap (mk-inj t1 g1) g2 h2
    ≡
    mk-gap t1 g1 (mk-proj g2 h2)
lem-mk-gap (‼ P1) g1 g2 (⁇ P2 l) = refl
lem-mk-gap ϵ (some l’ .(‘ _) .∗) g2 (⁇ P l) = refl
lem-mk-gap (‼ P) g1 (some l .∗ .(‘ _)) ϵ = refl
lem-mk-gap ϵ (some l’ .(‘ _) .∗) (some l .∗ .(‘ _)) ϵ = refl



seq-assoc : ∀ {T1 T2 T3 T4 T5 T6}
  → (c1 : Cast T1 T2)
  → (g1 : GapT T2 T3)
```

```
  → (c2 : Cast T3 T4)
  → (g2 : GapT T4 T5)
    → (c3 : Cast T5 T6)
  → seq (seq c1 g1 c2) g2 c3 ≡ seq c1 g1 (seq c2 g2 c3)


seq-m-assoc : ∀ {T1 T2 T3 T4 T5 T6}
  → (c1 : Body T1 T2)
  → (g1 : GapP T2 T3)
  → (c2 : Body T3 T4)
  → (g2 : GapP T4 T5)
  → (c3 : Body T5 T6)
  → seq-m (seq-m c1 g1 c2) g2 c3 ≡ seq-m c1 g1 (seq-m c2 g2 c3)


seq-m-assoc (⊥ l1) g1 m2 g2 m3 = refl
seq-m-assoc (‘ m1) g1 m2 g2 m3 with check-gap g1
seq-m-assoc (‘ m1) .(some l _ _) m2 g2 m3 | bad ¬P⌣Q l = refl
seq-m-assoc (‘ m1) g1 (⊥ l2) g2 m3 | good P⌣Q = refl
seq-m-assoc (‘ m1) g1 (‘ m2) g2 m3 | good P⌣Q with check-gap g2
seq-m-assoc (‘ m1) g1 (‘ m2) .(some l _ _) m3 | good P⌣Q | bad ¬P⌣Q l = refl
seq-m-assoc (‘ m1) g1 (‘ m2) g2 (⊥ l3) | good P⌣Q | good P⌣Q’ = refl
seq-m-assoc (‘ B̂) g1 (‘ B̂) g2 (‘ B̂) | good ⌣B | good ⌣B = refl
seq-m-assoc (‘ (c₁ ⇛ c₂)) g1 (‘ (c₃ ⇛ c₄)) g2 (‘ (c₅ ⇛ c₆)) | good ⌣⇛ | good ⌣⇛
  rewrite seq-assoc c₅ (g-dom g2) c₃ (g-dom g1) c₁
    | seq-assoc c₂ (g-cod g1) c₄ (g-cod g2) c₆
  = refl
seq-m-assoc (‘ (c₁ ⊗̂ c₂)) g1 (‘ (c₃ ⊗̂ c₄)) g2 (‘ (c₅ ⊗̂ c₆)) | good ⌣⊗ | good ⌣⊗
  rewrite seq-assoc c₁ (g-car g1) c₃ (g-car g2) c₅
    | seq-assoc c₂ (g-cdr g1) c₄ (g-cdr g2) c₆
```

```
  = refl


seq-assoc id∗ g1 id∗ g2 id∗ = refl
seq-assoc id∗ g1 id∗ g2 (↷ h m t) with (mk-proj g2 h)
seq-assoc id∗ g1 id∗ g2 (↷ h m t) | ⁇ P l = refl
seq-assoc id∗ g1 (↷ h m t) g2 id∗ = refl
seq-assoc id∗ g1 (↷ h m t) g2 (↷ h₁ m₁ t₁) = refl
seq-assoc (↷ h m t) g1 id∗ g2 id∗ with (mk-inj t g1)
seq-assoc (↷ h m t) g1 id∗ g2 id∗ | (‼ P) = refl
seq-assoc (↷ h1 m1 t1) g1 id∗ g2 (↷ h3 m3 t3)
  rewrite lem-mk-gap t1 g1 g2 h3
  = refl
seq-assoc (↷ h1 m1 t1) g1 (↷ h2 m2 t2) g2 id∗ = refl
seq-assoc (↷ h1 m1 t1) g1 (↷ h2 m2 t2) g2 (↷ h3 m3 t3)
  rewrite seq-m-assoc m1 (mk-gap t1 g1 h2) m2 (mk-gap t2 g2 h3) m3
  = refl


⨟-assoc : ∀ {T1 T2 T3 T4}
  → (c1 : Cast T1 T2)
  → (c2 : Cast T2 T3)
  → (c3 : Cast T3 T4)
  → (c1 ⨟ c2) ⨟ c3 ≡ c1 ⨟ (c2 ⨟ c3)
⨟-assoc c1 c2 c3 = seq-assoc c1 none c2 none c3


open import R.BlameStrategies Label using (BlameStrategy; LazyDBS)
open BlameStrategy LazyDBS using (Injectable)


open import S.Values Label Injectable Cast
```

```
open import Error
  using (Error; return; raise; _>>=_; _>=>_; >>=-return; >>=-assoc; >=>-assoc; >=>->>=)
open import Data.Unit using (tt)


CastResult : Type → Set
CastResult T = Error Label (Value T)


⟦_⟧t : ∀ {P T}
  → Tail P T
  → Value (‘ P)
  ---
  → CastResult T
⟦ !! P ⟧t v = return (dyn (same P) v)
⟦ ϵ   ⟧t v = return v


proxy : ∀ {P1 P2}
  → Value (‘ P1)
  → PreBody P1 P2
  ---
  → Value (‘ P2)
proxy v B̂ = v
proxy (lam⟨ c1 ⇒ c2 ⟩ e E) (c3 ⇏ c4)    = lam⟨ c3 ⨟ c1 ⇒ c2 ⨟ c4 ⟩ e E
proxy (cons⟨ c1 ⊗ c2 ⟩ v1 v2) (c3 ⊗̂ c4) = cons⟨ c1 ⨟ c3 ⊗ c2 ⨟ c4 ⟩ v1 v2


⟦_⟧m : ∀ {P1 P2}
  → Body P1 P2
  → Value (‘ P1)
```

```
  → CastResult (‘ P2)

⟦ ⊥ l ⟧m v = raise l

⟦ ‘ m ⟧m v = return (proxy v m)


⟦_⟧h : ∀ {T P}

  → Head T P

  → Value T

  → CastResult (‘ P)

⟦ ϵ      ⟧h v = return v

⟦ ?? P2 l ⟧h (dyn (same P1) v) = ⟦ seq-m (‘ id-m P1) (some l (‘ P1) (‘ P2)) (‘ id-m P2) ⟧m v


⟦_⟧ : ∀ {T1 T2}

  → Cast T1 T2

  → Value T1

  ---

  → CastResult T2

⟦ id*     ⟧ v = return v

⟦ ↷ h m t ⟧ v = ⟦ h ⟧h v >>= ⟦ m ⟧m >>= ⟦ t ⟧t


mutual

  lem-id-m : ∀ {P}

    → (v : Value (‘ P))

    ----------------------------

    → proxy v (id-m P) ≡ v

  lem-id-m {B} v = refl

  lem-id-m {S ⇒ T} (lam⟨ c ⇒ d ⟩ e E)  rewrite identity^l c | identity^r d = refl

  lem-id-m {S ⊗ T} (cons⟨ c ⊗ d ⟩ v u) rewrite identity^r c | identity^r d = refl
```

```
  lem-id : ∀ {T}
    → (v : Value T)
    ----------------------------
    → ⟦ id T ⟧ v ≡ return v
  lem-id {*} v = refl
  lem-id {‘ P} v rewrite lem-id-m v = refl


lem1 : ∀ {T1 T2 T3 T4}
  → (l  : Label)
  → (c1 : Cast T1 T2)
  → (c2 : Cast T3 T4)
  → (c1 ⨟ ⌈ T2 ⟹[ l ] T3 ⌉) ⨟ c2 ≡ seq c1 (some l T2 T3) c2
lem1 {T2 = T2} {T3 = T3} l c1 c2
  rewrite sym (seq-assoc c1 none (id _) (some l _ _) (id T3))
    | identityʳ c1
    | seq-assoc c1 (some l _ _) (id T3) none c2
    | identityˡ c2
  = refl


lem-seq-m : ∀ {P1 P2 T P3 P4}
  → (m1 : Body P1 P2)
  → (t1 : Tail P2 T)
  → (h2 : Head T  P3)
  → (m2 : Body P3 P4)
  → (∀ v →
       ⟦ seq-m m1 (mk-gap t1 none h2) m2 ⟧m v
         ≡
       (⟦ m1 ⟧m >=> ⟦ t1 ⟧t >=> ⟦ h2 ⟧h >=> ⟦ m2 ⟧m) v)
```

```
lem-seq : ∀ {T1 T2 T3}
  → (c1 : Cast T1 T2)
  → (c2 : Cast T2 T3)
  → ∀ v
  --------------------
  → ⟦ c1 ⨟ c2 ⟧ v ≡ ⟦ c1 ⟧ v >>= ⟦ c2 ⟧


lem-seq-m (⊥ l1) t1 h2 m2 v = refl
lem-seq-m (` m1) (!! P) (?? Q l) m2 v with (` P) ⌣? (` Q)
lem-seq-m (` m1) (!! P) (?? Q l) m2 v | no ¬p = refl
lem-seq-m (` m1) (!! P) (?? Q l) (⊥ l2) v | yes P⌣Q = refl
lem-seq-m (` B̂) (!! .B) (?? .B l) (` B̂) v | yes ⌣B = refl
lem-seq-m (` (c2 ⇛ d2)) (!! (S1 ⇒ T1)) (?? (S2 ⇒ T2) l) (` (c3 ⇛ d3))
          (lam⟨  c1 ⇒ d1 ⟩ e E) | yes ⌣⇒
  rewrite sym (seq-assoc c3 none ⌈ S2 ⟹[ l ] S1 ⌉ none (c2 ⨟ c1))
  | lem1 l c3 (c2 ⨟ c1) | lem1 l (d1 ⨟ d2) d3
  = cong₂ (λ c d → return (lam⟨ c ⇒ d ⟩ e E))
          (seq-assoc c3 _ c2 _ c1)
          (sym (seq-assoc d1 _ d2 _ d3))
lem-seq-m (` (c2 ⊗̂ d2)) (!! .(_ ⊗ _)) (?? .(_ ⊗ _) l) (` (c3 ⊗̂ d3))
          (cons⟨ c1 ⊗ d1 ⟩ v u) | yes ⌣⊗
  rewrite lem1 l (c1 ⨟ c2) c3 | lem1 l (d1 ⨟ d2) d3
  = cong₂ (λ c d → return (cons⟨ c ⊗ d ⟩ v u))
          (sym (seq-assoc c1 _ c2 _ c3))
          (sym (seq-assoc d1 _ d2 _ d3))
lem-seq-m (` m1) ϵ ϵ (⊥ l2) v = refl
lem-seq-m (` B̂) ϵ ϵ (` B̂) v = refl
lem-seq-m (` (c2 ⇛ d2)) ϵ ϵ (` (c3 ⇛ d3)) (lam⟨ c1 ⇒ d1 ⟩ e E)
```

```
  = cong₂ (λ c d → return (lam⟨ c ⇒ d ⟩ e E))
          (seq-assoc c3 none c2 none c1)
          (sym (seq-assoc d1 none d2 none d3))
lem-seq-m (‘ (c2 ⊗̂ d2)) ϵ ϵ (‘ (c3 ⊗̂ d3)) (cons⟨ c1 ⊗ d1 ⟩ v u)
  = cong₂ (λ c d → return (cons⟨ c ⊗ d ⟩ v u))
          (sym (seq-assoc c1 none c2 none c3))
          (sym (seq-assoc d1 none d2 none d3))


lem-seq id* c2 v
  rewrite identityˡ c2
  = refl
lem-seq (↷ h1 m1 t1) id* v
  rewrite identityʳ (↷ h1 m1 t1)
  | mk-inj-none t1
  | >>=-return (⟦ h1 ⟧h v >>= ⟦ m1 ⟧m >>= ⟦ t1 ⟧t)
  = refl
lem-seq (↷ h1 m1 t1) (↷ h2 m2 t2) v with ⟦ h1 ⟧h v
lem-seq (↷ h1 m1 t1) (↷ h2 m2 t2) v | raise l   = refl
lem-seq (↷ h1 m1 t1) (↷ h2 m2 t2) v | return v’
  rewrite >=>->>= (⟦ m1 ⟧m v’ >>= ⟦ t1 ⟧t) (⟦ h2 ⟧h >=> ⟦ m2 ⟧m) ⟦ t2 ⟧t
          | >=>->>= (⟦ m1 ⟧m v’ >>= ⟦ t1 ⟧t) ⟦ h2 ⟧h ⟦ m2 ⟧m
  = cong (_>>= ⟦ t2 ⟧t) (lem-seq-m m1 t1 h2 m2 v’)


H : CastADT Injectable
H = record
    { Cast = Cast
    ; id  = id
    ; ⌈_⌉ = ⌈_⌉
```

```
    ; _⨟_ = _⨟_
    ; ⟦_⟧ = ⟦_⟧
    ; lem-id = λ T v → lem-id v
    ; lem-seq = lem-seq
    }


eq-¬∪ : ∀ {T1 T2}
  → (l : Label)
  → ¬ (T1 ∪ T2)
  → (v : Value T1)
  ---
  → ⟦ ⌈ T1 ⟹[ l ] T2 ⌉ ⟧ v
      ≡
    raise l
eq-¬∪ {∗} {∗} l ¬S∪T v = ⊥-elim (¬S∪T ∗∪∗)
eq-¬∪ {∗} {` P} l ¬S∪T v = ⊥-elim (¬S∪T (∗∪P P))
eq-¬∪ {` P} {∗} l ¬S∪T v = ⊥-elim (¬S∪T (P∪∗ P))
eq-¬∪ {` P} {` Q} l ¬S∪T v with (` P) ∪? (` Q)
eq-¬∪ {` P} {` Q} l ¬S∪T v | yes p = ⊥-elim (¬S∪T p)
eq-¬∪ {` P} {` Q} l ¬S∪T v | no ¬p = refl


eq-∗∗ : ∀ l
  → (v : Value ∗)
  → ⟦ ⌈ ∗ ⟹[ l ] ∗ ⌉ ⟧ v
      ≡
    return v
eq-∗∗ l v = refl
```

```
eq-P∗ : ∀ {P}
  → (l : Label)
  → (v : Value (‘ P))
  → ⟦ ⌈ (‘ P) ⟹[ l ] ∗ ⌉ ⟧ v
      ≡
    return (dyn (same P) v)
eq-P∗ l v rewrite lem-id-m v = refl


eq-∗P : ∀ Q P l v
  → ⟦ ⌈ ∗   ⟹[ l ] ‘ Q ⌉ ⟧ (dyn (same P) v)
      ≡
    ⟦ ⌈ ‘ P ⟹[ l ] ‘ Q ⌉ ⟧ v
eq-∗P Q P l v with ⟦ seq-m (‘ id-m P) (some l (‘ P) (‘ Q)) (‘ id-m Q) ⟧m v
eq-∗P Q P l v | raise  l’ = refl
eq-∗P Q P l v | return v’ rewrite lem-id-m v’ = refl


open import S.LazyDCastADT Label


H-LazyD : LazyD H
H-LazyD =
  record
    { eq-¬⌣ = eq-¬⌣
    ; eq-∗∗ = eq-∗∗
    ; eq-P∗ = eq-P∗
    ; eq-∗P = eq-∗P
    ; eq-B = λ l v → refl
    ; eq-⇒ = λ T21 T22 T11 T12 {S} {T} l {Γ} c₁ c₂ e E → refl
    ; eq-⊗ = λ T21 T22 T11 T12 {S} {T} l c₁ c₂ v1 v2 → refl
```

```
    }
```

Appendix E

Major Theorems

```
module Contributions
  (Label : Set)
  where

open import Types using (Ground; Same)
open import Variables using (∅)
open import Terms Label using (_ ⊢ _)
open import Observables Label
open import R.Machine Label using ()
  renaming (Evalo to EvaloC)
open import S.Machine Label using ()
  renaming (Evalo to EvaloS)
open import S.CastADT Label using (CastADT)
open import S.LazyDCastADT Label using (LazyD)
open import S.LazyUDCastADT Label using (LazyUD)

open import Bisimulation.BisimulationProof Label using (correctness-l; correctness-r)
open import Bisimulation.LazyDApplyCast Label using ()
  renaming (lem-⟦_⟧ to lem-⟦_⟧-D)
open import Bisimulation.LazyUDApplyCast Label using ()
  renaming (lem-⟦_⟧ to lem-⟦_⟧-UD)

open import R.BlameStrategies Label using (BlameStrategy; LazyDBS; LazyUDBS)
open import CastRepresentations.LazyDHypercoercions Label using ()
  renaming (H to LazyDH; H-LazyD to LazyDH-LazyD)
```

```
open import CastRepresentations.LazyUDHypercoercions Label using ()
  renaming (H to LazyUDH; H-LazyUD to LazyUDH-LazyUD)
open import CastRepresentations.LazyUDCoercionsInNormalForm Label using ()
  renaming (S to LazyUDS; S-LazyUD to LazyUDS-LazyUD)
open import CastRepresentations.LazyUDThreesomes Label using ()
  renaming (C to LazyUDT; C-LazyUD to LazyUDT-LazyUD)


-- For all implementations of CastADT C, If C is LazyD
-- then C is correct (evalS(C,e) = o if and only if evalD(e) = o)


theorem-LazyD-CastADT-correct-part-1 : ∀ {T}
  → (C : CastADT Same)
  → (isLazyD : LazyD C)
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloS Same C e o
  ---
  → EvaloC LazyDBS e o
theorem-LazyD-CastADT-correct-part-1 C isLazyD prf
  = correctness-r LazyDBS C (lem-⟦_⟧-D C isLazyD) prf


theorem-LazyD-CastADT-correct-part-2 : ∀ {T}
  → (C : CastADT Same)
  → (lazyd : LazyD C)
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloC LazyDBS e o
  ---
```

```
  → EvaloS Same C e o
theorem-LazyD-CastADT-correct-part-2 C lazyd prf
  = correctness-l LazyDBS C (lem-⟦_⟧-D C lazyd) prf


-- For all implementations of CastADT C, If C is LazyUD
-- then C is correct (evalS(C,e) = o if and only if evalUD(e) = o)


theorem-LazyUD-CastADT-correct-part-1 : ∀ {T}
  → (C : CastADT Ground)
  → (lazyd : LazyUD C)
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloS Ground C e o
  ---
  → EvaloC LazyUDBS e o
theorem-LazyUD-CastADT-correct-part-1 C lazyd prf
  = correctness-r LazyUDBS C (lem-⟦_⟧-UD C lazyd) prf


theorem-LazyUD-CastADT-correct-part-2 : ∀ {T}
  → (C : CastADT Ground)
  → (lazyd : LazyUD C)
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloC LazyUDBS e o
  ---
  → EvaloS Ground C e o
theorem-LazyUD-CastADT-correct-part-2 C lazyd prf
  = correctness-l LazyUDBS C (lem-⟦_⟧-UD C lazyd) prf
```

```
-- Lazy D Hypercoercions

LazyDHypercoercionIsCorrect-1 : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloS Same LazyDH e o
  ---
  → EvaloC LazyDBS e o
LazyDHypercoercionIsCorrect-1
  = theorem-LazyD-CastADT-correct-part-1 LazyDH LazyDH-LazyD

LazyDHypercoercionIsCorrect-2 : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloC LazyDBS e o
  ---
  → EvaloS Same LazyDH e o
LazyDHypercoercionIsCorrect-2
  = theorem-LazyD-CastADT-correct-part-2 LazyDH LazyDH-LazyD

-- Lazy UD Hypercoercions

LazyUDHypercoercionIsCorrect-1 : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloS Ground LazyUDH e o
  ---
```

```
  → EvaloC LazyUDBS e o
LazyUDHypercoercionIsCorrect-1
  = theorem-LazyUD-CastADT-correct-part-1 LazyUDH LazyUDH-LazyUD


LazyUDHypercoercionIsCorrect-2 : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloC LazyUDBS e o
  ---
  → EvaloS Ground LazyUDH e o
LazyUDHypercoercionIsCorrect-2
  = theorem-LazyUD-CastADT-correct-part-2 LazyUDH LazyUDH-LazyUD


-- Lazy UD CoercionsInNormalForm


LazyUDCoercionsInNormalFormIsCorrect-1 : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloS Ground LazyUDS e o
  ---
  → EvaloC LazyUDBS e o
LazyUDCoercionsInNormalFormIsCorrect-1
  = theorem-LazyUD-CastADT-correct-part-1 LazyUDS LazyUDS-LazyUD


LazyUDCoercionsInNormalFormIsCorrect-2 : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloC LazyUDBS e o
```

```
  ---
  → EvaloS Ground LazyUDS e o
LazyUDCoercionsInNormalFormIsCorrect-2
  = theorem-LazyUD-CastADT-correct-part-2 LazyUDS LazyUDS-LazyUD


-- Lazy UD Threesomes


LazyUDThreesomeIsCorrect-1 : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloS Ground LazyUDT e o
  ---
  → EvaloC LazyUDBS e o
LazyUDThreesomeIsCorrect-1
  = theorem-LazyUD-CastADT-correct-part-1 LazyUDT LazyUDT-LazyUD


LazyUDThreesomeIsCorrect-2 : ∀ {T}
  → {e : ∅ ⊢ T}
  → {o : Observable T}
  → EvaloC LazyUDBS e o
  ---
  → EvaloS Ground LazyUDT e o
LazyUDThreesomeIsCorrect-2
  = theorem-LazyUD-CastADT-correct-part-2 LazyUDT LazyUDT-LazyUD
```

Appendix F

Other Modules

F.1 Chain

```
module Chain where


open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong)


data Chain {A : Set} (Link : A → A → Set) : A → A → Set where
  []  : ∀ {a} → Chain Link a a
  _::_ : ∀ {a b c} → Link a b → Chain Link b c → Chain Link a c


_++_ : ∀ {A Link} → {a b c : A} → Chain Link a b → Chain Link b c → Chain Link a c
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)


++-assoc : ∀ {A Link} → {a b c d : A}
  → (xs : Chain Link a b)
  → (ys : Chain Link b c)
  → (zs : Chain Link c d)
  → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc []       ys zs = refl
++-assoc (x :: xs) ys zs = cong (x ::_) (++-assoc xs ys zs)


++-identity^l : ∀ {A Link} → {a b : A}
  → (xs : Chain Link a b)
  → [] ++ xs ≡ xs
```

```
++-identityˡ xs = refl


++-identityʳ : ∀ {A Link} → {a b : A}
  → (xs : Chain Link a b)
  → xs ++ [] ≡ xs
++-identityʳ []       = refl
++-identityʳ (x :: xs) = cong (x ::_) (++-identityʳ xs)
```

F.2 Error Monad

```
module Error where


open import Relation.Binary.PropositionalEquality using (_≡_; refl)


infixl 30 _>>=_
infixl 30 _>=>_


data Error (Label : Set) (A : Set) : Set where
  raise  : (l : Label) → Error Label A
  return : (v : A) → Error Label A


Handler : ∀ (Label A B : Set) → Set
Handler Label A B = A → Error Label B


_>>=_ : ∀ {L A B}
  → Error L A
  → Handler L A B
  → Error L B
return x >>= h = h x
```

```
raise l >>= h = raise l


>>=-assoc : ∀ {L A B C}
  → (r : Error L A)
  → (f : Handler L A B)
  → (g : Handler L B C)
  → (r >>= f) >>= g
    ≡
    r >>= λ v → (f v >>= g)
>>=-assoc (return v) f g = refl
>>=-assoc (raise l) f g = refl


>>=-return : ∀ {L A}
  → (r : Error L A)
  → r >>= return ≡ r
>>=-return (return v) = refl
>>=-return (raise l) = refl


_>=>_ : ∀ {L A B C}
  → Handler L A B
  → Handler L B C
  → Handler L A C
(f >=> g) x = f x >>= g


>=>-assoc : ∀ {L A B C D}
  → (f : Handler L A B)
  → (g : Handler L B C)
  → (h : Handler L C D)
```

```
  → (∀ x → ((f >=> g) >=> h) x ≡ (f >=> (g >=> h)) x)
>=>-assoc f g h x = >>=-assoc (f x) g h


>=>->>= : ∀ {L A B C}
  → (r : Error L A)
  → (f : Handler L A B)
  → (g : Handler L B C)
  → r >>= (f >=> g) ≡ (r >>= f) >>= g
>=>->>= (raise l)  f g = refl
>=>->>= (return v) f g = refl


drop-after-raise : ∀ {L A B C}
  → (r : Error L A)
  → (l : L)
  → (f : Handler L B C)
  → (r >>= λ _ → raise l) >>= f
    ≡
    (r >>= λ _ → raise l)
drop-after-raise (raise l') l f = refl
drop-after-raise (return v) l f = refl


>>=-extensionality : ∀ {L A B}
  → (r : Error L A)
  → {f g : Handler L A B}
  → (f≡g : ∀ x → f x ≡ g x)
  → (r >>= f) ≡ (r >>= g)
>>=-extensionality (raise l)  f≡g = refl
>>=-extensionality (return v) f≡g = f≡g v
```